1 Hashing

Einleitung

Eine sehr naive Herangehensweise zur Implementierung eines Wörterbuchs ist die Benutzung eines (hinreichend grossen) unsortierten Arrays, in dem jeweils an eine freie Position eingefügt wird. Zur Suche eines Schlüssels werden dann im schlimmsten Fall $\Omega(n)$ viele Operationen benötigt, was natürlich völlig inakzeptabel ist. Etwas besser ist die Benutzung eines sortierten Arrays, bei dem die Suche im schlimmsten Fall mit $\mathcal{O}(\log n)$ vielen Operationen auskommt. Allerdings hat nun das Einfügen im schlimmsten Fall eine lineare Laufzeit, was erneut nicht hinnehmbar ist. Wir werden aber im Folgenden sehen, dass ein Array durchaus eine effiziente Realisierung eines Wörterbuchs sein kann, wenn die Positionen der zu verwaltenden Schlüssel im Array geschickt gewählt werden.

HASHING

Hashing benutzt ein Array T der Grösse m für ein eine hinreichend grosse Zahl $m \in \mathbb{N}$, und die Positionen seien von 0 bis $m-1$ nummeriert. Das Array T wird als *Hashtabelle* bezeichnet. Wie früher gehen wir davon aus, dass die zu verwaltenden Schlüssel aus $\mathcal{K} \subseteq \mathbb{N}_0$ stammen. Die Position eines Schlüssels k in T ergibt sich nun aus dem Wert von k selbst. Dazu wird eine *Hashfunktion* $h : \mathcal{K} \to \{0, 1, \ldots, m-1\}$ benutzt, die dem Schlüssel k seine *Hashadresse* $h(k)$ zuweist. Man beachte, dass im Allgemeinen $|\mathcal{K}| \gg m$ ist. Wir müssen also damit rechnen, dass es verschiedene Schlüssel $k, k' \in \mathcal{K}$ mit $h(k) = h(k')$ gibt, d.h., dass *Kollisionen* auftreten. Leider sind Kollisionen sogar recht wahrscheinlich, wie das folgende Theorem zeigt.

HASHTABELLE

HASHFUNKTION

HASHADRESSE

KOLLISION

Theorem 1 (Geburtstagsparadoxon)**.** *In einem Raum mit 23 Personen ist die Wahrscheinlichkeit, dass mindestens zwei dieser Personen am gleichen Tag Geburtstag haben, grösser als* 50%*.*

GEBURTSTAGS-PARADOXON

> **Beweis.** Nehmen wir vereinfachend an, dass keine Person am 29. Februar geboren wurde, dann gibt es 365 Möglichkeiten für den Geburtstag. Dann erhalten wir
>
> $$\begin{aligned} &P(\text{“Mindestens zwei Personen haben am gleichen Tag Geb.”}) && (1)\\ &= 1 - P(\text{“Alle Personen haben an verschiedenen Tagen Geb.”}) && (2)\\ &= 1 - \left(\frac{365}{365}\right) \cdot \left(\frac{364}{365}\right) \cdot \cdots \cdot \left(\frac{343}{365}\right) \approx 0.5073 > 0.5. && (3) \end{aligned}$$
>
> *Anmerkung:* Die Aussage gilt noch immer, wenn wir auch den 29. Februar als möglichen Geburtstag zulassen. ∎

In Bezug auf Hashing lässt sich das Geburtstagsparadoxon wie folgt interpretieren: Ist eine Hashtabelle der Grösse 365 gegeben und verteilen sich die Schlüssel aus $\mathcal{K}$ gleichmässig auf die Positionen $0, 1, \ldots, 364$, dann ist es schon bei 23 zufällig ausgewählten Schlüsseln wahrscheinlicher, dass eine Kollision auftritt, als dass keine auftritt. An die Hashfunktion stellen wir nun die folgenden Anforderungen:

ANFORDERUNGEN AN DIE HASHFUNKTION

1) Die Hashfunktion sollte gut “streuen”, d.h. die Schlüssel aus $\mathcal{K}$ möglichst gleichmässig auf die Positionen der Hashtabelle aufteilen.
2) Kollisionen müssen effizient auflösbar sein.

Wir werden im Folgenden Hashfunktionen vorstellen, die diese Anforderungen erfüllen.

Zur Wahl der Hashfunktion

DIVISIONS-REST-METHODE

Da wir $\mathcal{K} \subseteq \mathbb{N}_0$ angenommen haben, bietet sich die sog. *Divisions-Rest-Methode* an, die einem Schlüssel $k \in \mathcal{K}$ die Adresse $h(k) = k \bmod m$ zuweist. Für die Effizienz ist die Wahl der Grösse m der Hashtabelle entscheidend. Sehr schlecht ist die Wahl von $m = 2^i$ für ein $i \in \mathbb{N}$, also die Wahl einer Zweierpotenz. Dann entspricht nämlich $k \bmod m$ exakt den letzten i Bits von k, und alle anderen Bits werden ignoriert. Grundsätzlich ist es nicht empfehlenswert, m als gerade Zahl zu wählen: Dann nämlich ist $h(k)$ genau dann gerade, wenn k gerade ist. Dies ist negativ, da das letzte Bit von k einen Sachverhalt repräsentieren kann (z.B. wenn Personendatendaten verwaltet werden und das letzte Bit das Geschlecht der entsprechenden Person repräsentiert). Praktisch immer gut ist die Wahl von m als Primzahl.

MULTIPLIKATIVE METHODE

Eine weitere Methode ist die sog. *multiplikative Methode*. Dabei wird der Schlüssel k mit einer irrationalen Zahl multipliziert und der ganzzahlige Anzahl abgeschnitten. Dies liefert eine Zahl aus $[0, 1)$ zurück. Man kann zeigen, dass für die Schlüssel $\{1, \ldots, n\}$ die entstehenden Zahlen ziemlich gleichmässig über $[0, 1)$ verteilt sind [**?**]. Die beste Verteilung wird erreicht, wenn als irrationale Zahl das Inverse des goldenen Schnitts

$$\phi^{-1} = \frac{\sqrt{5} - 1}{2} \approx 0.61803 \tag{4}$$

gewählt wird. Wir erhalten mit dieser Methode dann die Hashfunktion

$$h(k) = \Big\lfloor m\big(k\phi^{-1} - \lfloor k\phi^{-1} \rfloor\big) \Big\rfloor. \tag{5}$$

Untersuchungen verschiedener Hashfunktionen haben gezeigt, dass von diesen die Divisions-Rest-Methode die Schlüssel im Durchschnitt am besten verteilt [**?**]. Zudem ist die Adresse leicht berechenbar. Im Folgenden nehmen wir also an, dass die Hashadresse mittels der Divisions-Rest-Methode bestimmt wird sofern nicht anders genannt; dabei werde die Grösse m der Hashtabelle als Primzahl gewählt.

Perfektes und universelles Hashing

Ist die Menge $K \subseteq \mathcal{K}$ *aller* zu speichernder Schlüssel im Voraus bekannt und gilt zusätzlich $|K| \leq m$, dann können offenbar alle Schlüssel kollisionsfrei gespeichert werden (wenn die Hashfunktion h als Bijektion zwischen K und einer $|K|$-elementigen Teilmenge von $\{0, \ldots, m-1\}$ gewählt wird). Diese Situation wird als *perfektes Hashing* bezeichnet. In der Praxis tritt sie natürlich höchst selten auf. Eher muss die gegenteilige Situation befürchtet werden, in der viele Schlüssel auf die gleiche Hashadresse abgebildet werden. Andererseits tritt diese Situation nur bei unglücklicher Wahl der Hashfunktion auf. Haben wir eine Menge $\mathcal{H}$ möglicher Hashfunktionen zur Verfügung und wählen wir zu Beginn des Einfügevorgangs einmalig eine Hashfunktion zufällig aus $\mathcal{H}$, dann führt eine "schlecht" gewählte Schlüsselmenge nicht notwendigerweise zu vielen Kollisionen. Wir werden im Folgenden diese Idee präzisieren.

PERFEKTES HASHING

UNIVERSELLES HASHING

Definition 2 (Universelles Hashing)**.** *Sei $\mathcal{H} \subseteq \{h : \mathcal{K} \to \{0, \ldots, m-1\}\}$ eine Klasse von Hashfunktionen. $\mathcal{H}$ heisst* universell*, falls*

$$\forall x, y \in \mathcal{K} \text{ mit } x \neq y : \frac{|\{h \in \mathcal{H} : h(x) = h(y)\}|}{|\mathcal{H}|} \leq \frac{1}{m} \tag{6}$$

gilt.

Mit anderen Worten: Ist $\mathcal{H}$ eine universelle Klasse von Hashfunktionen und wird für feste Schlüssel $x, y \in \mathcal{K}$ die Hashfunktion h zufällig aus $\mathcal{H}$ gewählt (und bleibt über alle Einfügevorgänge gleich), dann ist die Wahrscheinlichkeit für eine Kollision höchstens $1/m$. Seien nun H eine (nicht notwendigerweise universelle) Menge von Hashfunktion, $x, y \in \mathcal{K}$ zwei Schlüssel, $h \in H$ eine Hashfunktion und $Y \subseteq \mathcal{K}$ eine Menge von Schlüsseln. Definieren wir

$$\delta(x, y, h) = \begin{cases} 1 & \text{falls } h(x) = h(y) \text{ und } x \neq y \\ 0 & \text{sonst} \end{cases} \tag{7}$$

$$\delta(x, Y, h) = \sum_{y \in Y} \delta(x, y, h) \tag{8}$$

$$\delta(x, y, H) = \sum_{h \in H} \delta(x, y, h), \tag{9}$$

dann ist H genau dann universell, wenn $\forall x, y \in \mathcal{K}$ mit $x \neq y : \delta(x, y, H) \leq |\mathcal{H}|/m$ gilt. Wir nutzen diese Tatsache für den Beweis des folgenden Satzes.

Theorem 3. *Sei $\mathcal{H}$ eine universelle Klasse von Hashfunktionen. Eine zufällig gewählte Funktion $h \in \mathcal{H}$ verteilt im Erwartungswert eine beliebige Folge von Schlüsseln aus $\mathcal{K}$ so gleichmässig wie nur möglich auf die verfügbaren Plätze.*

Beweis. Wie wir gesehen haben, wird die Hashfunktion h initial zufällig aus $\mathcal{H}$ gewählt und bleibt über alle Einfügevorgänge gleich. Sei $S \subseteq \mathcal{K}$ die Menge der bereits in der Hashtabelle gespeicherten Schlüssel. Wird ein Schlüssel x neu eingefügt, dann beträgt die erwartete Anzahl von Kollisionen

$$\begin{aligned} \mathbb{E}_{\mathcal{H}}\big[\delta(x, S, h)\big] &= \sum_{h \in \mathcal{H}} \delta(x, S, h)/|\mathcal{H}| && (10) \\ &= \frac{1}{|\mathcal{H}|} \sum_{h \in \mathcal{H}} \sum_{y \in S} \delta(x, y, h) && (11) \\ &= \frac{1}{|\mathcal{H}|} \sum_{y \in S} \sum_{h \in \mathcal{H}} \delta(x, y, h) && (12) \\ &= \frac{1}{|\mathcal{H}|} \sum_{y \in S} \delta(x, y, \mathcal{H}) && (13) \\ &\leq \frac{1}{|\mathcal{H}|} \sum_{y \in S} |\mathcal{H}|/m = |S|/m. && (14) \end{aligned}$$

Dabei gelten die Gleichungen (10) aufgrund der Definition des Erwartungswerts, (11) und (13) aufgrund der Definitionen von $\delta(x, S, h)$ bzw. $(\delta(x, y, \mathcal{H}))$, und die Ungleichung (14) gilt wegen der vorherigen Charakterisierung universeller Hashklassen. ∎

Bisher haben wir lediglich gesehen, dass universelle Hashklassen angenehme Eigenschaften haben, wir haben aber noch keine universellen Hashklassen selbst gesehen. Glücklicherweise existieren diese und können sogar einfach konstruiert werden, wie der folgende Satz zeigt:

Theorem 4. *Seien p eine Primzahl und $\mathcal{K} = \{0, \ldots, p-1\}$. Definieren wir für $a \in \{1, \ldots, p-1\}$ und $b \in \{0, \ldots, p-1\}$ die Hashfunktion*

Existenz perfekter Hashklass

$$h_{ab} : \mathcal{K} \to \{0, \ldots, m-1\} \qquad (15)$$
$$k \mapsto \big((ak + b) \bmod p\big) \bmod m,$$

dann ist $\mathcal{H} = \{h_{ab} \mid 1 \leq a \leq p-1, 0 \leq b \leq p-1\}$ eine universelle Hashklasse. ∎

Hashverfahren mit Verkettung der Überläufer

Wir haben gesehen, dass wir immer mit Kollisionen rechnen müssen, wenn die Schlüssel im Vorfeld nicht bekannt sind (d.h., wenn perfektes Hashing nicht anwendbar ist). Es gibt nun verschiedene Strategien zur Kollisionsbehandlung. Eine naheliegende Variante besteht darin, an jeder Tabellenposition nicht einen einzigen Schlüssel, sondern eine Liste von Schlüsseln zu speichern, die alle die gleiche Hashadresse besitzen. Man spricht dann auch von einer *direkten Verkettung der Überläufer*. Daneben existiert auch noch eine *separate Verkettung der Überläufer*. Bei dieser wird ein Schlüssel k direkt in der Hashtabelle gespeichert, wenn die Position $h(k)$ frei ist; alle weiteren Elemente, die mit k kollidieren, werden in einer Liste der Überläufer ausserhalb der Hashtabelle verwaltet. Wir werden diese Strategie im folgenden nicht weiter betrachten.

Direkte und Separate Verkettung

Werden die Überläufer direkt verkettet, dann können die Wörterbuchoperationen sehr leicht wie folgt implementiert werden.

Suchen SEARCH(k). Durchsuche die an Tabellenposition $h(k)$ gespeicherte Liste und gib "Wahr" zurück, falls sie k enthält. Wird das Ende der Liste erreicht, ohne dass k gefunden wird, dann ist k nicht in der Hashtabelle vorhanden, also gib "Falsch" zurück.

Einfügen INSERT(k). Prüfe, ob die an der Tabellenposition $h(k)$ gespeicherte Liste den Schlüssel k enthält. Falls nein, füge k am Ende dieser Liste ein.

Löschen DELETE(k). Durchsuche die an der Tabellenposition $h(k)$ gespeicherte Liste nach k. Stoppt die Suche erfolgreich bei einem Listenelement, das k speichert, dann entferne dieses Listenelement.

Die Laufzeit der Operationen hängt offenbar von der Grösse der Hashtabelle und der Anzahl der gespeicherten Schlüssel ab. Hat eine Hashtabelle Grösse m und werden derzeit n Schlüssel verwaltet, dann wird $\alpha = n/m$ als *Belegungsfaktor* bezeichnet. Für eine genaue Laufzeitabschätzung beobachten wir, dass die Laufzeit des Einfügens und des Löschens durch die Zeit zur erfolglosen bzw. erfolgreichen Suche nach einem Schlüssel dominiert werden. Daher genügt es, die erwartete Anzahl betrachteter Listeneinträge

Belegungsfaktor

- C'_n bei erfolgloser Suche, sowie
- C_n bei erfolgreicher Suche

zu analysieren. Bei einer erfolglosen Suche nach einem Schlüssel k müssen alle Elemente der an Position $h(k)$ gespeicherten Liste betrachtet werden. Die Anzahl betrachteter Elemente ist gleich der Länge der entsprechenden Liste. Da die durchschnittliche Listenlänge $\alpha = n/m$ beträgt, erhalten wir

Erfolglose Suche

$$C'_n = \alpha. \qquad (16)$$

FOLGREICHE SUCHE

Ist die Suche dagegen erfolgreich und befindet sich der gesuchte Schlüssel k im i-ten Eintrag in der entsprechenden Liste, dann werden genau i Listeneinträge betrachtet. Um den Erwartungswert von i zu bestimmen, analysieren wir die Zeit zur erfolgreichen Suche des j-ten eingefügten Schlüssels. Vereinfachend nehmen wir an, dass Schlüssel grundsätzlich ans Listenende eingefügt werden, und dass niemals Schlüssel gelöscht wurden. Vor dem Einfügen des j-ten Schlüssels betrug die durchschnittliche Listenlänge $(j-1)/m$. Also betrachtet die erfolgreiche Suche nach diesem Schlüssel im Erwartungswert $1+(j-1)/m$ Listeneinträge. Für jedes $j \in \{1,\dots,n\}$ beträgt die Wahrscheinlichkeit, dass k als j-tes eingefügt wurde, genau $1/n$. Also ist

$$C_n = \frac{1}{n}\sum_{j=1}^{n}\big(1+(j-1)/m\big) = 1+\frac{n-1}{2m} \approx 1+\frac{\alpha}{2}. \tag{17}$$

VERBESSERUNG

Das Verfahren hat noch Potential zur Verbesserung. Wir haben der Einfachheit halber angenommen, dass ein Schlüssel immer ans Ende der entsprechenden Liste eingefügt wird. Halten wir dagegen die Listen sortiert (d.h., fügen wir einen Schlüssel gleich an der korrekten Position der sortierten Reihenfolge ein), dann verbessert dies die Zeit zur erfolglosen Suche. Die Zeit zur erfolgreichen Suche verschlechtert sich nicht, lediglich ihre Analyse wird komplizierter.

VOR- UND NACHTEILE

Vorteile der Strategie sind, dass Belegungsfaktoren $\alpha > 1$ problemlos möglich sind, d.h., dass mehr Schlüssel verwaltet werden können als die Tabelle Plätze hat. Ausserdem ist das Entfernen von Schlüsseln unkompliziert und ohne negative Konsequenzen möglich. Negativ ist der erhöhte Speicherplatzbedarf, denn für jeden Listeneintrag muss ein Zeiger auf den nachfolgenden Eintrag gespeichert werden.

Offenes Hashing

Anstatt die Überläufer in einer Liste zu verwalten, können wir sie auch direkt in der Hashtabelle speichern. Auf diese Weise entfällt der für die Listenzeiger benötigte Extraplatz. Um bei diesem *offenen* Hashing die Position der Überläufer zu finden, benutzen wir eine *Sondierungsfunktion* $s(j,k)$, wobei $j \in \{0,\dots,m-1\}$ und $k \in \mathcal{K}$ ein Schlüssel ist. Zur Bestimmung der Tabellenposition eines Schlüssels k betrachten wir die Tabellenpositionen entlang der *Sondierungsfolge*

SONDIERUNGSFUNKTION

SONDIERUNGSFOLGE

$$\big(h(k)-s(0,k)\big) \bmod m, \dots, \big(h(k)-s(m-1,k)\big) \bmod m. \tag{18}$$

Speichert ein Eintrag $\big(h(k)-s(j,k)\big) \bmod m$ den Schlüssel k, dann haben wir die Position von k gefunden und sind fertig (die übrigen Positionen der Folge müssen dann nicht mehr betrachtet werden). Ist der Eintrag frei, dann ist der Schlüssel k in der Tabelle nicht vorhanden und wir sind ebenfalls fertig. Weitersuchen müssen wir lediglich, wenn der Eintrag einen Schlüssel ungleich k enthält. In diesem Fall fahren wir analog mit dem Tabelleneintrag $\big(h(k)-s(j+1,k)\big) \bmod m$ fort. Die Wörterbuchoperationen können nun wie folgt implementiert werden.

SUCHEN

SEARCH(k). Betrachte die Tabelleneinträge gemäss der Sondierungsfolge (18). Speichert ein Tabelleneintrag den Schlüssel k, gib "Wahr" zurück und brich die Suche ab. Wird ein leerer Tabelleneintrag gefunden oder ist k in der gesamten Sondierungsfolge nicht vorhanden, gib "Falsch" zurück.

EINFÜGEN

INSERT(k). Führe eine Suche nach dem Schlüssel k durch. Ist der Schlüssel k nicht vorhanden, füge k an die erste freie Position der Sondierungsfolge ein.

DELETE(k). Führe eine Suche nach dem Schlüssel k durch. Falls k gefunden wird, markiere die Position von k mit einem speziellen Gelöscht-Flag. Die Position darf nicht einfach als frei markiert werden, da ansonsten die Suche nach einem Schlüssel nicht mehr funktioniert (werden z.B. viele Schlüssel $k_1, \ldots, k_l$ mit der gleichen Hashadresse eingefügt und wird dann k_1 gelöscht, dann wäre die erste Position der Sondierungsfolge ein freier Tabellenplatz und die Suche würde direkt terminieren). Das Gelöscht-Flag kann z.B. durch einen speziellen Schlüssel repräsentiert werden, der dann natürlich nicht mehr regulär eingefügt werden darf. Wird bei der Suche ein als gelöscht markiertes Feld gefunden, muss die Suche regulär mit der nächsten Position der Sondierungsfolge fortgesetzt werden. Neue Schlüssel dürfen aber natürlich auf Positionen eingefügt werden, die als gelöscht markiert sind.

Löschen

Nachdem wir die Realisierung der Wörterbuchoperationen beschrieben haben, müssen wir nun überlegen, was geeignete Sondierungsfunktionen sind. Auf jeden Fall sollte die Sondierungsfolge (18) alle Tabellenpositionen berücksichtigen, d.h., sie sollte eine Permutation von $\{0, \ldots, m-1\}$ sein. Wir beschreiben nun einige Sondierungsstrategien, die diese Eigenschaft besitzen.

Lineares Sondieren

Lineares Sondieren: $s(j,k) = j$**.** Die zugehörige Sondierungsfolge ist

$$h(k) \bmod m, \big(h(k)-1\big) \bmod m, \ldots, \big(h(k)-m+1\big) \equiv \big(h(k)+1\big) \bmod m, \quad (19)$$

d.h., ausgehend von der Position $h(k)$ laufen wir in der Tabelle so lange nach links, bis eine freie Position gefunden wird. Wird die Position ganz links (mit Index 0) erreicht, fahren wir bei der Position ganz rechts (mit Index $m-1$) fort. Diese Sondierungsstrategie hat den Nachteil, dass auch bei geringen Belegunsfaktoren α schnell lange zusammenhängende Blöcke von belegten Tabellenplätzen entstehen. Um dies einzusehen, nehmen wir an, alle Tabellenpositionen i innerhalb eines Intervalls $[i_l, i_u] \subseteq \{0, \ldots, m\}$ wären bereits belegt. Wird nun ein Schlüssel k mit Hashadresse $h(k) \in [i_l, i_u]$ eingefügt, dann wird dieser direkt links vom genannten Bereich gespeichert (wenn $i_l > 0$ ist; für $i_l = 0$ wird der Schlüssel entsprechend möglichst weit rechts in der Tabelle gespeichert). Aus diesem Grund wachsen bereits bestehende lange Blöcke schneller als kurze Blöcke. Diese Eigenschaft führt auch zu langen Suchzeiten: Man kann zeigen, dass die erfolglose bzw. erfolgreiche Suche

$$C'_n \approx \frac{1}{2}\Big(1 + \frac{1}{(1-\alpha)^2}\Big) \quad \text{bzw.} \quad C_n \approx \frac{1}{2}\Big(1 + \frac{1}{1-\alpha}\Big) \quad (20)$$

viele Tabellenpositionen betrachten. Dies zeigt, wie ineffizient lineares Sondieren bei hohen Belegungsfaktoren ist: Für $\alpha = 0.95$ betrachtet eine erfolglose Suche im Durchschnitt mehr als 200 Tabelleneinträge!

Quadratisches Sondieren

Quadratisches Sondieren: $s(j,k) = (\lceil j/2 \rceil)^2 (-1)^j$**.** Ein Problem beim linearen Sondieren besteht darin, dass ähnliche Hashadressen auch ähnliche Sondierungsfolgen besitzen, was wie gesehen schnell zu langen zusammenhängenden Bereichen von belegten Positionen führt. Quadratisches Sondieren vermeidet dieses Problem, indem die Abstände zwischen der ursprünglichen Hashadresse und der entsprechenden Adresse in der Sondierungsfolge quadratisch wachsen. Zur oben angegebenen Sondierungsfunktion gehört die Sondierungsfolge

$$\begin{aligned} &h(k) \bmod m, \big(h(k)-1\big) \bmod m, \big(h(k)+1\big) \bmod m, \\ &\quad \big(h(k)-4\big) \bmod m, \big(h(k)+4\big) \bmod m, \big(h(k)-9\big) \bmod m, \ldots \end{aligned} \quad (21)$$

Man kann zeigen, dass (21) eine Permutation aller Tabellenpositionen $\{0, \ldots, m-1\}$ bildet, wenn m eine Primzahl ist und $m \equiv 3 \bmod 4$ erfüllt. Die erfolglose bzw. erfolgreiche Suche betrachten

$$C'_n \approx \frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{1-\alpha}\right) \quad \text{bzw.} \quad C_n \approx 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2} \tag{22}$$

viele Tabellenpositionen. Insbesondere betrachtet die erfolglose Suche bei einem Belegungsfaktor von $\alpha = 0.95$ nur ca. 22 Tabelleneinträge, was eine enorme Verbesserung gegenüber linearem Sondieren darstellt. Trotzdem hat diese Sondierungsstrategie einen Nachteil: Alle Schlüssel mit gleichen Hashadressen besitzen die gleiche Sondierungsfolge. Werden nun viele Schlüssel mit der gleichen Hashadresse eingefügt, dann führt dies zu vielen sogenannten *Sekundärkollisionen* entlang der Sondierungsfolge. Wir werden im Folgenden eine Strategie kennenlernen, bei der auch Schlüssel mit gleichen Hashadressen unterschiedliche Sondierungsfolgen haben können.

SEKUNDÄRKOLLISION

Double Hashing Sei $h'(k)$ eine zweite, von $h(k)$ unabhängige Hashfunktion. *Double Hashing* besitzt die Sondierungsfunktion $s(j,k) = j \cdot h'(k)$ mit der zugehörigen Sondierungsfolge

DOUBLE HASHING

$$h(k) \bmod m, \big(h(k) - h'(k)\big) \bmod m, \big(h(k) - 2h'(k)\big) \bmod m, \ldots \tag{23}$$

Die Wahl der zweiten Hashfunktion $h'(k)$ ist nichttrivial. Offenbar darf sie m nicht teilen und muss für alle Schlüssel k ungleich 0 sein. Man kann aber zeigen, dass alle genannten Anforderungen erfüllt werden, wenn m eine Primzahl ist und $h(k) = k \bmod m$ sowie $h'(k) = 1 + (k \bmod (m-2))$ gewählt werden. In diesem Fall betrachten die erfolglose bzw. erfolgreiche Suche

$$C'_n \approx \frac{1}{1-\alpha} \quad \text{bzw.} \quad C_n \approx 1 + \frac{\alpha}{2} + \frac{\alpha^3}{4} + \frac{\alpha^4}{15} - \frac{\alpha^5}{18} + \ldots < 2.5 \tag{24}$$

viele Tabellenpositionen.