FREIE UNIVERSITÄT BERLIN

Theoretische und Experimentelle Untersuchung von Kuckucks-Hashing

Bachelorarbeit
zur Erlangung des Grades Bachelor of Science
im Studiengang Informatik

Vorgelegt am 15. Januar 2013 von

Terese Haimberger

Erstgutachter	**Zweitgutachter**
Prof. Dr. Wolfgang Mulzer	Dr. Frank Hoffmann

Fachbereich Mathematik und Informatik

Eidesstattliche Erklärung

Ich versichere, dass ich die Bachelorarbeit selbständig verfasst, und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit hat keiner anderen Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass bei Verwendung von Inhalten aus dem Internet ich diese zu kennzeichnen und einen Ausdruck mit Angabe des Datums sowie der Internet-Adresse als Anhang der Bachelorarbeit anzugeben habe.

Terese Haimberger, Berlin, den 15. Januar 2013

Inhaltsverzeichnis

Zusammenfassung

Kuckucks-Hashing ist ein Hashverfahren, das einen amortisiert konstanten erwarteten Zeitaufwand für Einfügeoperationen und einen garantiert konstanten Zeitaufwand für Such- und Löschoperationen anbietet. Der Speicherplatzbedarf für eine Schlüsselmenge der Größe n ist dabei $2(1+\varepsilon)n$ für ein $\varepsilon > 0$. Die wesentliche Schwäche des Verfahrens ist, dass das Einfügen einer Schlüsselmenge mit Wahrscheinlichkeit $O(1/n)$ scheitert; es müssen neue Hashfunktionen gewählt und alle Schlüssel neu eingefügt werden.

Im ersten Teil unserer Arbeit stellen wir einen Kodierungsbeweis zur Laufzeit der Einfügeoperation und zur Wahrscheinlichkeit des Scheiterns vor, der zuerst von Mihai Pătraşcu skizziert wurde. Wir verallgemeinern den Beweis, so dass er für beliebige $\varepsilon > 0$, statt nur für $\varepsilon = 1$, funktioniert.

Im zweiten Teil führen wir Experimente durch, die einerseits die theoretischen Ergebnisse aus dem ersten Teil bestätigen und andererseits zeigen, dass es effiziente Hashfunktionen gibt, die beim Kuckucks-Hashing vergleichbare Eigenschaften aufweisen wie zufällige Funktionen; die durchschnittliche Anzahl der Schritte beim Einfügen und die Wahrscheinlichkeit des Scheiterns steigen nur geringfügig.

Kapitel 1

Einleitung

1.1 Motivation

Definition 1. Ein ***dynamisches Wörterbuch*** ist ein abstrakter Datentyp, der eine Menge von Schlüsseln repräsentiert und drei Operationen zur Verwaltung der Schlüssel anbietet:

`insert(`x`)` fügt den Schlüssel x in das Wörterbuch ein.

`delete(`x`)` entfernt den Schlüssel x aus dem Wörterbuch.

`lookup(`x`)` gibt genau dann `true` zurück, wenn x im Wörterbuch enthalten ist.

Eine typische Variante dieses Wörterbuchs repräsentiert nicht eine Menge von Schlüsseln, sondern eine Menge von Paaren (x, y) von Schlüsseln x und assoziierten Werten y. Bei dieser Variante gibt z. B. `lookup(`x`)` nicht mehr einen Booleschen Wert, sondern den mit x assoziierten Wert, zurück. Wir beschäftigen uns in dieser Arbeit nur mit dem Speichern und Verwalten von Schlüsseln, aber die vorgestellten Algorithmen, Sätze und Beweise lassen sich leicht modifizieren, so dass sie auch für die Variante funktionieren.

Die effizientesten Implementierungen von dynamischen Wörterbüchern basieren auf Hashverfahren wie Hashing mit Verkettung, lineares Sondieren und doppeltes Hashen, die Schlüssel in Tabellen speichern und Hashfunktionen verwenden, um schnell auf gespeicherte Schlüssel zuzugreifen und neue Schlüssel einzufügen.

Nachteil vieler gängigen Hashverfahren ist aber, dass die *erwartete* Laufzeit der Operationen `delete` und `lookup` konstant ist, die Laufzeit *im schlimmsten Fall* aber nicht. Bei Anwendungen wie Router, die selten neue Einträge zur Routingtabelle hinzufügen und viel öfter nach Einträgen suchen, wäre es wünschenswert, dass das Suchen auch im schlimmsten Fall in konstanter Zeit gelingt.

Genau diese Garantie liefert das 2001 von Pagh und Rodler[6] eingeführte Kuckucks-Hashing.

1.2 Kuckucks-Hashing

1.2.1 Beschreibung des Verfahrens

Im Folgenden sei U die Menge aller Schlüssel und $S \subseteq U$ eine Schlüsselmenge mit $|S| = n$.

Kuckucks-Hashing verwendet zwei Hashtabellen T_0 und T_1 der Größe $m \in \mathbb{N}$ mit zugehörigen Hashfunktionen $h_0, h_1 : U \to \{0, \ldots, m-1\}$. Jeder Schlüssel $x \in S$ kann an höchstens einer von zwei Stellen gespeichert werden: $T_0[h_0(x)]$ oder $T_1[h_1(x)]$. Beim Suchen und beim Entfernen von Schlüsseln brauchen wir also nur diese zwei Stellen zu betrachten:

procedure LOOKUP(x)
    return $T_0[h_0(x)] = x \ \vee \ T_1[h_1(x)] = x$
end procedure

Beim Entfernen ersetzen wir ggf. das Element durch $\perp$, wobei $\perp$ heißen soll, dass kein Element an der Stelle gespeichert ist.

procedure DELETE(x)
    if $T_0[h_0(x)] = x$ **then**
        $T_0[h_0(x)] \leftarrow \perp$
    else if $T_1[h_1(x)] = x$ **then**
        $T_1[h_1(x)] \leftarrow \perp$
end procedure

Richtig interessant wird das Verfahren erst beim Einfügen der Schlüssel. Sei $x \in S$ ein einzufügender Schlüssel, der noch nicht in den Hashtabellen gespeichert ist. Dann versuchen wir zuerst, x an der Stelle $T_0[h_0(x)]$ einzufügen. Falls diese Position schon durch einen Schlüssel y besetzt ist, gehen wir wie das Kuckucksweibchen vor: Wir werfen y hinaus und speichern unser „Ei“ x an der dadurch frei gewordenen Stelle. Nun versuchen wir, das „nestlose Ei“ y an seiner alternativen Stelle $T_1[h_1(y)]$ einzufügen. Falls auch diese Position durch einen Schlüssel z besetzt ist, werfen wir z hinaus und ersetzen es durch y. Dann machen wir von vorne mit dem Schlüssel z weiter, bis wir irgendwann eine freie Stelle finden, so dass der aktuelle Schlüssel ohne Hinauswerfen eines anderen eingefügt werden kann.

Es kann vorkommen, dass wir nie eine freie Stelle finden, z. B. wenn es drei Schlüssel $x, y, z \in S$ gibt mit $h_0(x) = h_0(y) = h_0(z)$ und $h_1(x) = h_1(y) = h_1(z)$. In solchen Fällen schieben wir die Schlüssel nur hin und her, ohne näher an das Ziel zu kommen. Um diese Endlosschleifen zu vermeiden, führen wir eine Konstante `MaxLoop` ein. Wenn die bisherige

```
procedure INSERT(x)
    if lookup(x) then return
    for i = 1 → MaxLoop do
        x ↔ T_0[h_0(x)]
        if x = ⊥ then return
        x ↔ T_1[h_1(x)]
        if x = ⊥ then return
    rehash()
    insert(x)
end procedure
```

Abbildung 1.1: Pseudocode zur Einfüge-Operation. Die Variable x speichert das aktuelle „nestlose Ei“ und $\leftrightarrow$ ist die Operation, die die Werte zweier Variablen tauscht.

Anzahl der Schleifendurchgänge diese Konstante überschreitet, brechen wir ab, bauen die Hashtabellen neu auf und versuchen erneut, den Schlüssel x einzufügen.

Ein ***Neuaufbau*** (engl. *rehash*) der Hashtabellen besteht im Wesentlichen aus drei Schritten: (1) Wähle neue Hashfunktionen h_0 und h_1, (2) lösche alle Schlüssel aus den Hashtabellen und (3) füge sie wieder anhand der neuen Hashfunktionen ein.

Definition 2. Wir sagen, dass eine Schlüsselmenge S der Größe n ***erfolgreich eingefügt*** werden kann, wenn für jede beliebige Nummerierung der Schlüssel $x_1, x_2, \ldots, x_n \in S$ gilt: die Operationsfolge

$$\texttt{insert}(x_1), \texttt{insert}(x_2), \ldots, \texttt{insert}(x_n)$$

führt zu keinem Neuaufbau der Tabellen, und die Operation `lookup`(x) liefert nach dem Ausführen dieser Operationsfolge für jedes $x \in S$ den Wert `true`, und `false` für alle Elemente $x \in U \setminus S$. Umgangssprachlich: Wir können alle Schlüssel $x \in S$ in beliebiger Reihenfolge eingefügen.

Es sei nebenbei bemerkt, dass es nicht möglich ist, dass das Einfügen in manchen Reihenfolgen gelingt und in anderen nicht. Wir werden nämlich im folgenden Kapitel sehen, dass der Erfolg des Einfügens von statischen Eigenschaften der Schlüsselmenge und der Hashfunktionen abhängt, nicht vom dynamischen Zustand der Hashtabellen. Wir könnten also eine äquivalente Definition formulieren, die nur die Existenz einer erfolgreichen Reihenfolge fordert.

1.2.2 Laufzeit

Wie schon erwähnt, garantiert Kuckucks-Hashing auch im schlimmsten Fall, dass die Laufzeit von `lookup` und `delete` konstant ist. Das Verfahren kann zwar keine solche Garantie für `insert` liefern, aber die amortisierte erwartete Laufzeit von `insert` ist nachweislich konstant.

1.2.3 Varianten

In dieser Arbeit beschäftigen wir uns nicht nur mit dem herkömmlichen Kuckucks-Hashing-Verfahren, sondern auch mit zwei Varianten von Kuckucks-Hashing: die „Tabellen-Variante“ (engl. *d-ary cuckoo hashing*) und die „Behälter-Variante“. Um diese Varianten zu beschreiben, führen wir zunächst einige Begriffe ein, die wir immer wieder im Laufe der Arbeit verwenden.

Definition 3. Wir nennen die Positionen der Hashtabellen auch ***Behälter***. Die ***Behältergröße*** b ist durch die Anzahl der Schlüssel definiert, die in einem Behälter gespeichert werden können und die ***Tabellengröße*** m ist die Anzahl der Behälter pro Hashtabelle. Die ***Kapazität*** c der Hashtabelle ergibt sich aus der Multiplikation $d \cdot m \cdot b$, wobei d die Anzahl der Hashtabellen ist. Mit anderen Worten: Die Kapazität ist die maximale Anzahl von Schlüsseln, die in die Hashtabellen passen könnten.

Sei nun S eine Schlüsselmenge der Größe n, die in den Hashtabellen gespeichert ist. Dann ist der aktuelle ***Lastfaktor*** l der Hashtabellen gegeben durch n/c.

Eine Schwäche des herkömmlichen Kuckucks-Hashing-Verfahrens ist, dass die Einfügezeiten sehr schnell steigen, wenn der aktuelle Lastfaktor zu nahe des Lastfaktors 0.5 kommt. Um diese Schwäche zu beheben, wurden zwei Verallgemeinerungen des Verfahrens vorgeschlagen:

Tabellen-Variante Wir verwenden nicht unbedingt zwei Hashtabellen, sondern eine beliebige aber feste Anzahl $d \geq 2$ von Hashtabellen und entsprechend vielen Hashfunktionen[4].

Behälter-Variante Wir erlauben Behältergrößen $b \geq 1$, wobei b wieder eine beliebige, aber feste Zahl ist[3].

Eine weitere Variante, die wir in dieser Arbeit nicht genauer betrachten, ist das von Kirsch und Mitzenmacher[5] vorgeschlagene Verfahren, das Kuckucks-Hashing mit Hilfe einer Schlange deamortisiert. Arbitman et al. haben bewiesen, dass die Laufzeit von `insert` bei dieser Variante auch im schlimmsten Fall konstant ist[2]. Diese Variante ist besonders interessant, weil Kuckucks-Hashing sonst für manche Anwendungen wegen der eventuellen Neuaufbauten der Hashtabelle nicht in Frage kommen würde.

1.3 Beitrag der Arbeit

Im theoretischen Teil dieser Arbeit stellen wir eine Ausarbeitung eines Kodierungsbeweises zur Laufzeit von `lookup` vor, der von Mihai Pătraşcu in seinem Blog skizziert wurde[7]. In seinem Beweis, setzt Pătraşcu voraus, dass die Größe der Hashtabellen $m = 2n$ ist (was dem Lastfaktor 0.25 entspricht). Wir zeigen aber, dass der Beweis genau so gut für Hashtabellen der Größe $m = (1 + \varepsilon)n$ funktioniert.

Der Beweis soll einerseits als Alternative zum Beweis von Pagh und Rodler dienen und andererseits dazu motivieren, Kodierungsbeweise (die das Beweisen einer Aussage auf das Finden eines Kodierungsalgorithmus reduzieren) auch für andere Sätze aus der Informatik zu verwenden.

Im experimentellen Teil der Arbeit verwenden wir eine eigene Implementierung von Kuckucks-Hashing, um Experimente durchzuführen. Diese Experimente bestätigen, dass sich die Laufzeit von `lookup` sowohl unter idealen als auch unter realistischen Umständen so verhält, wie wir es nach der Theorie erwarten würden.

Kapitel 2

Theoretische Untersuchung

2.1 Kuckucksgraph

Im Folgenden seien T_0 und T_1 Hashtabellen der Größe m, die wir als Mengen $[m] := \{0, \ldots, m-1\}$ von Positionen darstellen, U die Menge aller Schlüssel und $h_0, h_1 : U \to [m]$ Hashfunktionen.

Definition 4. Sei $S \subseteq U$ eine Schlüsselmenge. Als ***Kuckucksgraph*** bezeichnen wir den bipartiten Graphen $G(S) := (T_0, T_1, \{((h_0(x), h_1(x)), x) \mid x \in S\})$, der für jeden Schlüssel $x \in S$ eine mit x markierte ungerichtete Kante zwischen den beiden möglichen Positionen des Schlüssels enthält.

Ein ***Weg*** der Länge $k \in \mathbb{N}$ im Kuckucksgraphen ist eine Folge

$$(y_0, h_a(y_0) = h_a(y_1), y_1, h_b(y_1) = h_b(y_2), y_2, h_a(y_2) = h_a(y_3), \ldots, y_k, h_c(y_k))$$

von Schlüsseln und Knoten mit $a, c \in \{0, 1\}$, $b = 1 - a$ und $y_i \in S$, $i \in \{0, \ldots, k\}$. Das Einfügen eines Schlüssels x entspricht dem Traversieren eines Weges im Graphen, der mit x beginnt und mit einer zuvor freien Stelle endet.

Ein ***Pfad*** ist ein Weg mit paarweise verschiedenen Schlüsseln und Knoten.

Ein ***Kreis*** ist ein Weg, wobei der letzte Knoten gleich dem ersten ist ($h_c(y_k) = h_a(y_1)$) und sonst alle Schlüssel und Knoten paarweise verschieden sind.

Wir nennen einen Kuckucksgraphen ***gültig***, wenn jede Zusammenhangskomponente des Graphen höchstens einen Kreis enthält.

Bemerkung 1. Sei $G = (V, E)$ ein zusammenhängender Graph. Dann ist

$\lvert V\rvert = \lvert E\rvert + 1$,	gdw. G keine Kreise enthält,
$\lvert V\rvert = \lvert E\rvert$,	gdw. G genau einen Kreis enthält,
$\lvert V\rvert < \lvert E\rvert$,	gdw. G mehrere Kreise enthält.

Satz 1. *Sei $S \subseteq U$ eine Schlüsselmenge und $G(S)$ der zugehörige Kuckucksgraph. Wir können genau dann alle Schlüssel $x \in S$ erfolgreich einfügen (siehe Def. 2), wenn $G(S)$ ein gültiger Kuckucksgraph ist.*

Beweis. Wir beweisen zuerst die Kontraposition der Hinrichtung:

$$G(S) \text{ ungültig} \overset{!}{\Longrightarrow} \text{ das Einfügen scheitert.}$$

Nach Bemerkung 1 finden wir in jedem ungültigen Kuckucksgraphen eine Zusammenhangskomponente, die mehr Kanten als Knoten enthält. Mit anderen Worten, wir finden eine Zusammenhangskomponente, für die die Anzahl der einzufügenden Schlüssel echt größer ist als die Anzahl der Speicherplätze. Mindestens einer der Schlüssel kann nicht erfolgreich eingefügt werden; das Einfügen scheitert.

Nun beweisen wir die Rückrichtung durch Induktion über S:

$$G(S) \text{ gültig} \overset{!}{\Longrightarrow} \text{ das Einfügen gelingt.}$$

Induktionsanfang: Für $S = \emptyset$ gilt die Aussage trivialerweise.

Induktionsvoraussetzung: Sei $S \subseteq U$ eine Schlüsselmenge, für die der Kuckucksgraph gültig ist. Dann können wir alle Schlüssel $x \in S$ erfolgreich einfügen.

Induktionsschritt: Sei $x \in U \setminus S$ ein Schlüssel, so dass $G(S \cup \{x\})$ gültig ist. Dann ist auch $G(S)$ gültig. Wir wissen also nach Induktionsvoraussetzung, dass wir alle Schlüssel $y \in S$ in jeder beliebigen Reihenfolge erfolgreich einfügen können. Um nun zu zeigen, dass wir auch ein letztes Element x erfolgreich einfügen können, zeigen wir, dass der beim Einfügen von x traversierte Weg $W(x)$ eine endliche Länge hat.

Fall 1 $W(x)$ enthält keine Kreise. Dann ist die Länge von $W(x)$ durch die Anzahl der Knoten begrenzt; d. h. wir finden nach endlich vielen Schritten einen freien Platz.

Fall 2 $W(x)$ enthält mindestens einen Kreis. Wir behaupten, dass dann $W(x)$ die

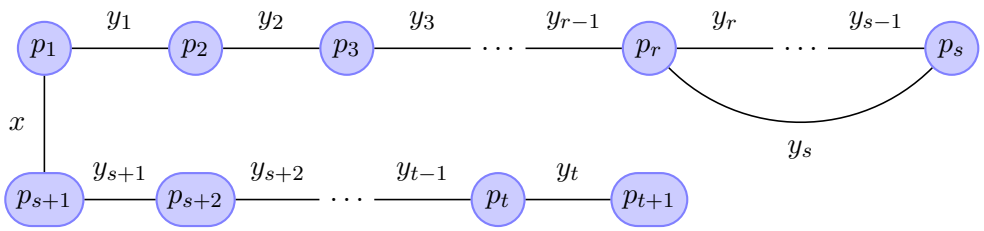


Abbildung 2.1: Untergraph des Kuckucksgraphen $G(S \cup \{x\})$ im 2. Fall des Induktionsschritts.

folgende Form hat:

$$(\underbrace{x, p_1, y_1, p_2, y_2, \ldots, p_r, y_r, \ldots, p_s, y_s, p_r}_{\text{Phase 1}},$$
$$\underbrace{y_{r-1}, p_{r-1}, y_{r-2}, p_{r-2}, \ldots, y_1, p_1}_{\text{Phase 2}},$$
$$\underbrace{x, p_{s+1}, y_{s+1}, p_{s+2}, \ldots, p_t, y_t, p_{t+1}}_{\text{Phase 3}})$$

mit $1 \leq r < s$, $s \leq t$, $p_1 = h_0(x)$, $p_{s+1} = h_1(x)$ und paarweise verschiedenen Positionen p_i. Um diesen Weg zu erklären, zerlegen wir ihn in drei Phasen:

Phase 1: Der Anfang des Weges (bis zum zweiten Auftreten der Position p_r) folgt direkt aus der Voraussetzung, dass $W(x)$ einen Kreis enthält.

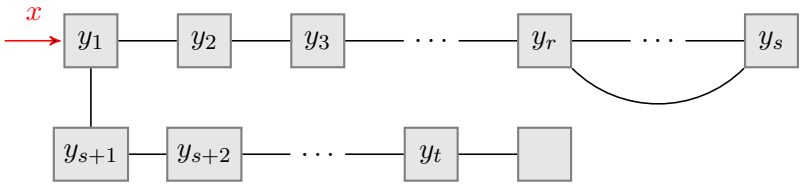
Seien $y_1, \ldots, y_s$ die Schlüssel, die vor dem Einfügen von x an den Stellen $p_1, \ldots, p_s$ gespeichert wurden. Beim Einfügen von x werden zunächst alle diese Schlüssel verschoben, so dass x in p_1 und y_i in p_{i+1} liegt, $1 \leq i < s$. Als Letztes wird der Schlüssel y_s an der Stelle p_r eingefügt.

Phase 2: Falls $r > 1$ ist, wird dadurch der Schlüssel y_{r-1} ein zweites Mal hinausgeworfen. Er muss also wieder an der ursprünglichen Stelle p_{r-1} eingefügt werden, was wiederum zum Hinauswerfen des Schlüssels y_{r-2} führt, usw. Irgendwann erreichen wir wieder die Position p_1 und ersetzen das dort eingefügte x durch y_1. Falls $r = 1$ ist, entfällt diese Phase; das x wird am Ende der 1. Phase direkt durch y_s ersetzt.

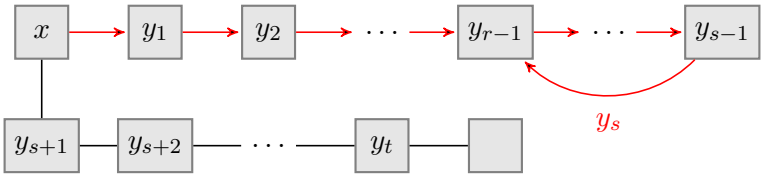
Phase 3: Nun muss das x ein zweites Mal eingefügt werden, dieses Mal in Tabelle B. Wenn die Position $p_{s+1} = h_1(x)$ frei ist, sind wir fertig ($t = s$); sonst wird ein Schlüssel y_{s+1} hinausgeworfen. Dieser Schlüssel muss wieder eingefügt werden, was ggf. einen Schlüssel y_{s+2} hinauswirft, usw.

Aus der Gültigkeit des Graphen $G(S \cup \{x\})$ folgt, dass die Zusammenhangskomponente, in der wir uns befinden, höchstens einen Kreis enthält. Da wir schon einen Kreis $(y_{r-1}, p_r, y_r, p_{r+1}, \ldots, p_s, y_s, p_r)$ gefunden haben, dürfen sonst keine Kreise auftauchen; d. h. alle Positionen p_i, $i > s$, müssen paarweise verschieden sein. Damit ist die Länge des letzten Teilweges $(x, p_{s+1}, y_{s+1}, p_{s+2}, \ldots)$ durch die Anzahl der Knoten beschränkt. Die Länge des gesamten Weges ist also endlich; das Einfügen von x gelingt.

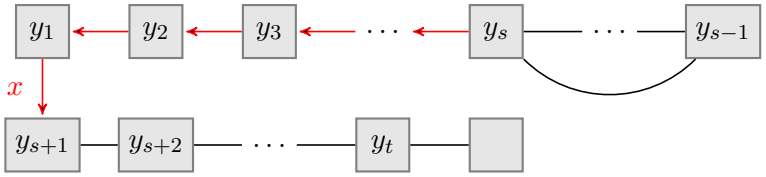
□



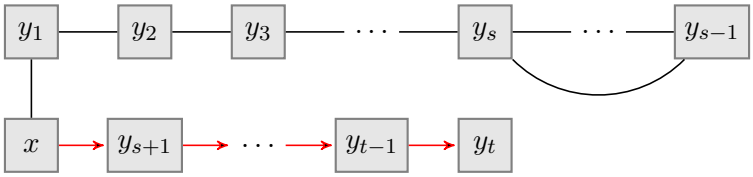
(a) Die Belegung der Speicherplätze vor dem Einfügen von x.



(b) Phase 1: Alle Schlüssel y_i, $1 \leq i < s$, werden nach rechts verschoben. Im nächsten Schritt muss y_s wieder eingefügt werden.



(c) Phase 2: Die Schlüssel y_i, $1 \leq i < r$, werden wieder nach links verschoben. Im nachsten Schritt muss x wieder eingefügt werden.



(d) Phase 3: Alle Schlüssel y_i, $s < i \leq t$, werden nach rechts verschoben. Der letzte Schlüssel y_t wird an einer freien Stelle eingefügt und wir sind fertig.

Abbildung 2.2: Einfügen eines Schlüssels im 2. Fall des Induktionsschritts.

2.2 Laufzeitanalyse

2.2.1 Suchen und Löschen

Da es für jeden Schlüssel nur zwei mögliche Positionen gibt, und da wir an jeder Position höchstens einen Schlüssel speichern, ist die Laufzeit der im 1. Kapitel beschriebenen Algorithmen zum Suchen und Löschen auch im schlimmsten Fall konstant.

Auch bei den Varianten, wo wir d Hashtabellen verwenden und in jedem Behälter b Schlüssel speichern, brauchen wir höchstens $d \cdot b$ Positionen zu betrachten. Die Laufzeit im schlimmsten Fall bleibt also konstant.

2.2.2 Einfügen

Im Folgenden sei $S \subseteq U$ die Menge der einzufügenden Schlüssel, $n = |S|$ die Anzahl dieser Schlüssel, $m = (1+\varepsilon)n$ mit $\varepsilon > 0$ die jeweilige Größe der Hashtabellen T_0 und T_1, und

$$\mathcal{H} := \{(f, g) \mid f, g : S \to [m]\}.$$

Wir setzen voraus, dass n eine Zweierpotenz ist und dass die Hashfunktionen $(h_0, h_1) \in \mathcal{H}$ zufällige Funktionen sind.

Wir zeigen mit Hilfe des von Mihai Pătraşcu beschriebenen Kodierungsbeweises[7], dass das Einfügen eines Schlüssels für alle $k \in \mathbb{N}$ mit Wahrscheinlichkeit $2^{-\Omega(k)}$ k Schritte benötigt und dass das Einfügen von S mit Wahrscheinlichkeit $O(1/n)$ scheitert (d. h. nicht ohne Neuaufbau der Hashtabellen gelingt). Dazu klären wir zuerst ein paar Begriffe.

Definition 5. Ein ***Ereignis*** ist eine Menge $\mathcal{E} \subseteq \mathcal{H}$ von Paaren von Hashfunktionen. Das Ereignis tritt ein, wenn $(h_0, h_1) \in \mathcal{E}$ ist.

Eine ***Kodierung*** einer Menge M ist eine injektive Funktion $c : M \to \{0,1\}^*$.

Bemerkung 2. Sei $c^* : \mathcal{H} \to \{0,1\}^*$ die Kodierung, die einfach alle Hashwerte der Reihe nach binär kodiert auflistet. Dann gilt $\forall (f,g) \in \mathcal{H} : |c^*(f,g)| = 2n \log m =: H$ und $|\mathcal{H}| = m^{2n} = 2^{\log(m^{2n})} = 2^H$.

Falls für ein Ereignis $\mathcal{E} \subseteq \mathcal{H}$ gilt, dass wir beim Eintreten des Ereignisses immer Δ Bits sparen können, d. h.

$$\exists \text{ Kodierung } c \;\; \forall (f,g) \in \mathcal{E} : \;\; |c(f,g)| = H - \Delta,$$

dann wissen wir, dass $\Pr[\mathcal{E}] \leq 2^{-\Delta}$ sein muss. Mit $H - \Delta$ Bits können wir nämlich höchstens $2^{H-\Delta}$ Paare von Funktionen kodieren, im Vergleich zu den 2^H Paaren, die wir sonst kodieren könnten.

$$\Pr[\mathcal{E}] = \frac{|\mathcal{E}|}{|\mathcal{H}|} \leq \frac{2^{H-\Delta}}{2^H} = 2^{-\Delta}$$

Lemma 1. *Sei $x \in S$ ein Schlüssel und $k \in \mathbb{N}$. Wenn das Ereignis $A_k := \{(f,g) \mid$ der zugehörige Kuckucksgraph $G(S)$ enthält einen Pfad der Länge $l = \Omega(k)$ als Untergraph, der mit x beginnt$\}$ eintritt, dann können wir $\Delta = \Omega(k)$ Bits sparen.*

Beweis. Wir kodieren die Hashfunktionen folgenderweise:

Bits	**Inhalt**
1	Ist der erste Knoten im Pfad $h_0(x)$ oder $h_1(x)$?
$O(\log l)$	die Zahl l ($\log l$ Einsen, gefolgt von einer 0 und l als Binärzahl)
$l \cdot \log n$	alle Schlüssel außer x, die auf dem Pfad liegen (der Reihe nach)
$l \cdot \log m$	alle Knoten des Pfades (der Reihe nach), bis auf den ersten[1]
$(n-l)2\log m$	die Hashwerte aller restlichen Schlüssel (inklusive x)

Statt die Knoten des Pfades zweimal zu kodieren (was die naive Kodierung c^* getan hätte), kodieren wir die Schlüssel und die Knoten des Pfades jeweils einmal. Wir sparen dadurch Bits, weil $m = (1+\varepsilon)n$ ist; d. h. wir brauchen zum Kodieren eines Schlüssels nur $\log n = \log m - \log(1+\varepsilon)$ Bits.

Sei nun H' die Anzahl der Bits, die wir für diese Kodierung brauchen. Dann ist

$$\begin{aligned}
\Delta &= H - H' \\
&= 2n\log m - (1 + O(\log l) + l \cdot \log n + l \cdot \log m + (n-l)2\log m) \\
&= (2n - l - l - 2(n-l))\log m - 1 - O(\log l) + l \cdot \log(1+\varepsilon) \\
&= l \cdot \log(1+\varepsilon) - O(\log l) - 1 \\
&= \Omega(l) = \Omega(k).
\end{aligned}$$

□

Satz 2. *Sei $S \subseteq U$ eine Menge von Schlüsseln, die schon erfolgreich eingefügt wurden, $x \in U \setminus S$ ein Schlüssel, der ohne Neuaufbau der Tabellen eingefügt werden kann, und $\mathcal{E}_k := \{(f,g) \mid$ das Einfügen von x benötigt k Schritte$\}$, $k \in \mathbb{N}$. Dann ist $\Pr[\mathcal{E}_k] \leq 2^{-\Omega(k)}$.*

Beweis. Sei $W(x)$ der beim Einfügen von x traversierte Weg. Wir definieren zwei Ereignisse:

$\mathcal{E}_1 := \{(f,g) \mid W(x)$ ist ein Pfad der Länge k $\}$
$\mathcal{E}_2 := \{(f,g) \mid W(x)$ ist ein Weg der Länge k, der mindestens einen Kreis enthält $\}$.

Es gilt: $\mathcal{E}_k = \mathcal{E}_1 \dot\cup \mathcal{E}_2$. Wir zeigen: (i) $\mathcal{E}_1 \subseteq A_k$ und (ii) $\mathcal{E}_2 \subseteq A_k$, wobei A_k wie im Lemma 1 definiert ist.

(i) Nehmen wir an, dass das Ereignis $\mathcal{E}_1$ eintritt. Dann ist der Weg $W(x)$ ein Pfad der Länge k, der mit x beginnt. Das Ereignis A_k tritt also auch ein.

[1] Wir brauchen den ersten Knoten $h_0(x)$ bzw. $h_1(x)$ hier nicht zu kodieren, weil er schon im folgenden Teil der Kodierung als Hashwert von x kodiert wird.

(ii) Nehmen wir an, dass das Ereignis $\mathcal{E}_2$ eintritt. Im Beweis zum Satz 1 haben wir bereits festgestellt, dass der Weg dann die folgende Form hat:

$$(x, p_1, y_1, p_2, y_2, \ldots, p_r, y_r, \ldots, p_s, y_s, p_r, \\ y_{r-1}, p_{r-1}, y_{r-2}, p_{r-2}, \ldots, y_1, p_1, \\ x, p_{s+1}, y_{s+1}, p_{s+2}, \ldots, p_t, y_t, p_{t+1})$$

mit $1 \leq r < s$, $s \leq t = k - r$, $p_1 = h_0(x)$, $p_{s+1} = h_1(x)$ und paarweise verschiedenen Positionen p_i. Dieser Weg enthält zwei Pfade, die mit x beginnen:

$$(x, p_1, y_1, p_2, y_2, \ldots, p_r, y_r, \ldots, y_{s-1}, p_s)$$
$$(x, p_{s+1}, y_{s+1}, p_{s+2}, \ldots, p_t, y_t, p_{t+1}).$$

Die Länge des 2. Pfades ist $t - s = k - r - s \geq k - 2s$. Wir finden also im Graphen $G(S \cup \{x\})$ einen Pfad der Länge $\max(s - 1, k - 2s) = \Omega(k)$, der mit x beginnt; d. h. das Ereignis A_k tritt ein.

Nun können wir $\Pr[\mathcal{E}_k]$ berechnen:

$$\begin{aligned} \Pr[\mathcal{E}_k] &\leq \Pr[\mathcal{E}_1] + \Pr[\mathcal{E}_2] && \text{(weil } \mathcal{E}_k = \mathcal{E}_1 \dot\cup \mathcal{E}_2 \text{ ist)} \\ &\leq \Pr[A_k] + \Pr[A_k] && \text{(nach (i) und (ii))} \\ &\leq 2^{-\Omega(k)} + 2^{-\Omega(k)} && \text{(nach Lemma 1 und Bemerkung 2)} \\ &\leq 2^{-\Omega(k)}. \end{aligned}$$

□

Korollar 1. Sei $x \in S$ ein Schlüssel, der ohne Neuaufbau der Tabellen eingefügt werden kann, und $k \in \mathbb{N}$ die Einfügezeit von x, d. h. die Anzahl der Schritte, die wir beim Einfügen von x benötigen. Dann ist die erwartete Einfügezeit $\mathrm{E}[k]$ konstant.

$$\mathrm{E}[k] = \sum_{k \in \mathbb{N}} k \Pr[\mathcal{E}_k] = \sum_{k \in \mathbb{N}} \frac{k}{2^{-\Omega(k)}} = O(1)$$

Satz 3. *Das Einfügen der Schlüsselmenge S scheitert mit Wahrscheinlichkeit $O(1/n)$.*

Beweis. Nehmen wir an, dass das Ereignis $\mathcal{E}_f := \{(f, g) \mid$ das Einfügen der Schlüsselmenge S scheitert, wenn die Hashfunktionen f und g verwendet werden$\}$ eintritt. Nach Satz 1 wissen wir, dass das Einfügen von S genau dann scheitert, wenn $G(S)$ ungültig ist, d. h. wenn mindestens eine Zusammenhangskomponente im Kuckucksgraphen mindestens zwei Kreise enthält. Wir betrachten den kleinsten zusammenhängenden Untergraphen von $G(S)$, der zwei Kreise enthält.

Fall 1 Es handelt sich um einen Kreis mit einer Sehne (siehe Abbildung 2.3).

Fall 2 Es handelt sich um zwei Kreise, die höchstens einen Knoten gemeinsam haben (siehe Abbildung 2.4).

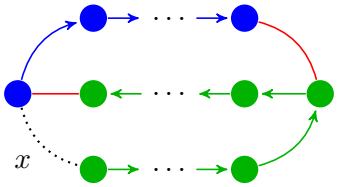


Abbildung 2.3: Der Untergraph im 1. Fall.

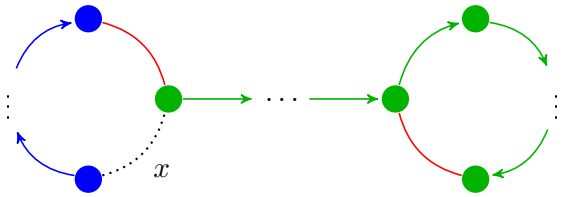


Abbildung 2.4: Der Untergraph im 2. Fall.

In beiden Fällen finden wir einen Schlüssel $x \in S$ und zwei Kanten (in den Abbildungen rot markiert), so dass wir durch das Entfernen dieser Kanten zwei Pfade (jeweils blau und grün markiert) erhalten, die mit x beginnen. Seien l_1 und l_2 die Längen der beiden Pfade und k die Größe des Untergraphen. Dann ist in beiden Fällen $l_1 + l_2 = k - 2$, d. h.

$$\max(l_1, l_2) = \Omega(k). \tag{2.1}$$

Die zwei zu entfernenden Kanten finden wir auch in sehr kleinen Untergraphen, z. B. im trivialen Fall $k = 2$, wo der Untergraph aus zwei Knoten und drei Kanten besteht. Es kann durchaus vorkommen, dass einer (oder, im Fall $k = 2$, sogar beide) der Pfade Länge 0 hat. Das stört aber nicht bei unseren Rechnungen, weil $\max(l_1, l_2) = \Omega(k)$ auf alle Fälle gilt.

Wir zeigen nun, wie wir mindestens $\log n - O(1)$ Bits beim Kodieren der Hashfunktionen sparen können.

1. Wir betrachten zuerst den Untergraphen ohne die entfernten Kanten. Nach (2.1) enthält dieser Untergraph mindestens einen Pfad der Länge $l = \Omega(k)$, der mit x beginnt. Wenn wir also den Untergraphen gemäß dem im Beweis zum Lemma 1 beschriebenen Schema kodieren und noch den Schlüssel x mit $\log n$ Bits kodieren, sparen wir $\Omega(k) - \log n$ Bits. (Der Schlüssel x muss hier extra kodiert werden, weil er nicht wie im Satz 2 aus dem Ereignis $\mathcal{E}_f$ hervorgeht.)

2. Die zwei entfernten Kanten kodieren wir mit jeweils $\log n + O(\log k)$ Bits ($\log n$ Bits, um die Kante zu identifizieren, und $O(\log k)$ Bits, um die zwei Endpunkte der Kante zu identifizieren). Mit der naiven Kodierung c^* hätten wir jeweils $2 \log m$ Bits gebraucht. Wir sparen also $2(2 \log m) - 2(\log n + O(\log k))$ Bits.

3. Als Letztes listen wir die verbleibenden Hashwerte auf, wodurch wir nichts gegenüber der naiven Kodierung sparen, aber auch nichts verlieren.

Sei nun H' die Anzahl der Bits, die wir für diese Kodierung brauchen. Dann ist

$$\begin{aligned}\Delta &= H - H' \\ &\geq \Omega(k) - \log n + 2(2\log m) - 2(\log n + O(\log k)) \\ &\geq \Omega(k) - \log n + 4\log n + 4(1+\varepsilon) - 2\log n - O(\log k) \\ &\geq \log n + \Omega(k) - O(\log k) + 4(1+\varepsilon) \\ &\geq \log n - O(1).\end{aligned}$$

Nach Bem. 2 ist die Wahrscheinlichkeit, dass diese Fälle auftreten, durch $2^{-\Delta}$ beschränkt. Damit bekommen wir für $\mathcal{E}_f$:

$$\Pr[\mathcal{E}_f] \leq 2^{-\Delta} \leq 2^{-(\log n - O(1))} = 2^{O(1)}(2^{\log n})^{-1} = 2^{O(1)}n^{-1} = O(1/n). \qquad \square$$

Im Korollar zum Satz 2 hatten wir gezeigt, dass die erwartete Einfügezeit eines Schlüssels $x \in S$ konstant ist unter der Voraussetzung, dass x überhaupt ohne Neuaufbau der Tabellen eingefügt werden kann. Dieses Ergebnis wenden wir im folgenden Korollar an.

Korollar 2. Die amortisierte erwartete Einfügezeit T_a eines Schlüssels ist konstant.

Beweis. Sei $\mathrm{E}[T(n)]$ die erwartete gesamte Einfügezeit für die Schlüsselmenge S. Wir betrachten zwei Fälle:

Fall 1 S kann erfolgreich eingefügt werden. Dann ist die erwartete Laufzeit nach dem oben genannten Korollar $n \cdot O(1) = O(n)$.

Fall 2 Mindestens ein Neuaufbau der Tabellen ist erforderlich. Dann müssen wir alle Schlüssel noch einmal einfügen, was wieder $\mathrm{E}[T(n)]$ Schritte benötigt. Dazu kommen noch $O(n)$ Schritte, weil wir möglicherweise fast alle Schlüssel einfügen, bevor wir zu einem Schlüssel kommen, der nicht erfolgreich eingefügt werden kann.

Zusammengefasst:

$$\begin{aligned}\mathrm{E}[T(n)] &\leq (1 - \Pr[\mathcal{E}_f])O(n) + \Pr[\mathcal{E}_f](\mathrm{E}[T(n)] + O(n)) \\ &\leq O(n) + \Pr[\mathcal{E}_f]\mathrm{E}[T(n)].\end{aligned}$$

Wir formen die Ungleichung um und bekommen

$$\mathrm{E}[T(n)] \leq \frac{O(n)}{1 - \Pr[\mathcal{E}_f]} \leq \frac{O(n)}{1 - O(1/n)} = O(n).$$

Also:

$$T_a = \frac{\mathrm{E}[T(n)]}{n} = \frac{O(n)}{n} = O(1).$$

$\square$

Anmerkung. Pătraşcu hat zum Beweis des 3. Satzes zuerst gezeigt, dass das Einfügen *eines* Schlüssels mit Wahrscheinlichkeit $O(1/n^2)$ zu einem Neuaufbau der Hashtabellen führt. Wir haben den Kodierungsalgorithmus leicht modifiziert, so dass wir den Satz direkt (und mit weniger Fallunterscheidungen) beweisen konnten.

Kapitel 3

Experimentelle Untersuchung

In diesem Kapitel bestimmen wir anhand von Experimenten, wie sich das Einfügen unter unterschiedlichen Bedingungen verhält. Wir interessieren uns insbesondere dafür, wieviele Kollisionen beim Einfügen eines Schlüssels auftreten und wie oft es vorkommt, dass ein Schlüssel nicht ohne Neuaufbau der Tabellen eingefügt werden kann.

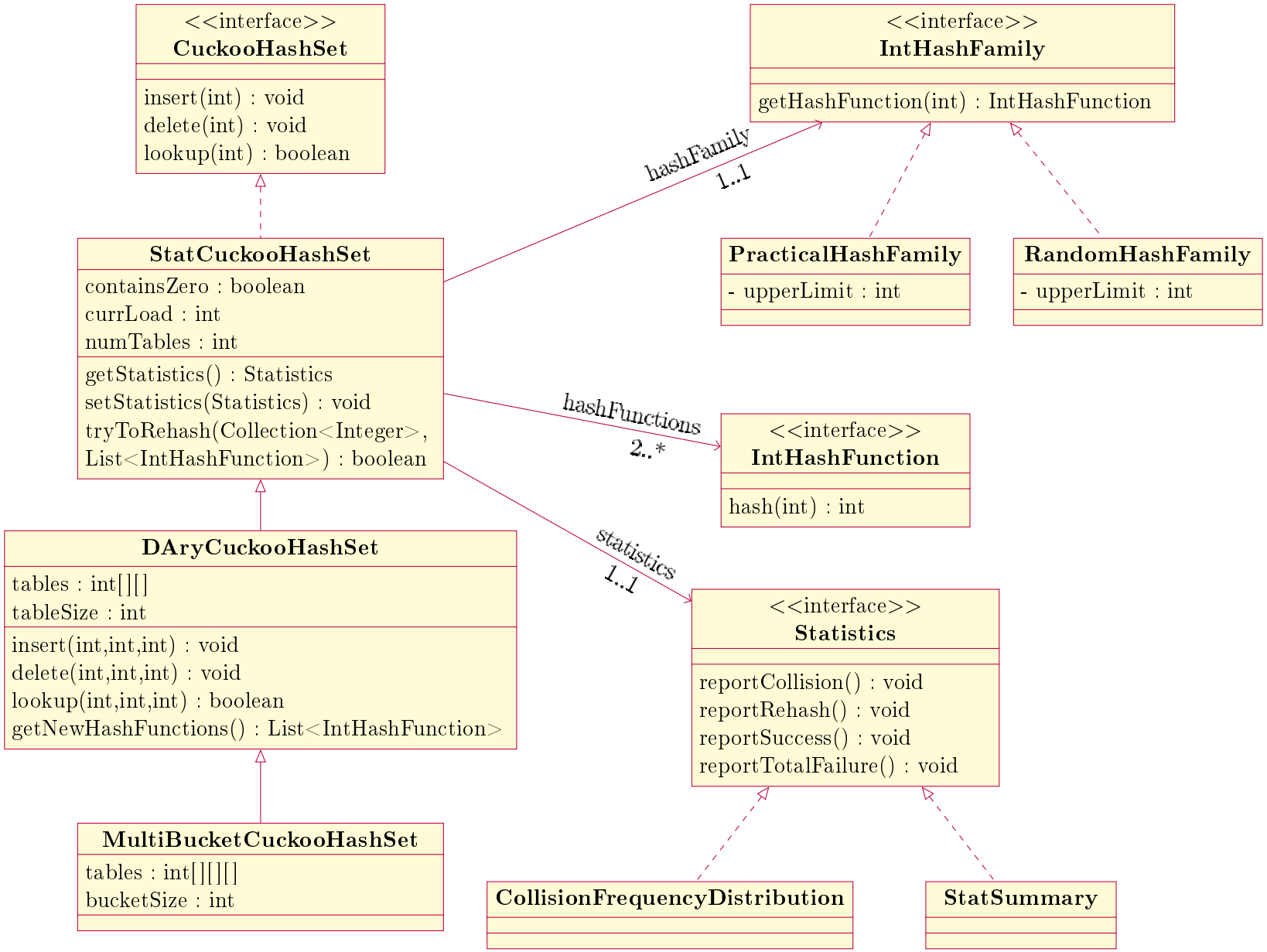
Definition 6. Im Rahmen der Experimente sprechen wir von einer ***Kollision***, wenn wir versuchen, einen Schlüssel an einer Stelle einzufügen, wo sich bereits ein anderer Schlüssel befindet. Die Anzahl der Kollisionen, die beim Einfügen eines Schlüssels x auftreten, entspricht also der Länge des beim Einfügen von x traversierten Weges im Kuckucksgraphen.

3.1 Implementierung von Kuckucks-Hashing

Wir implementieren die von `CuckooHashSet` spezifizierte Schnittstelle, die unter anderem die Methoden `delete`, `insert` und `lookup` bereitstellt. Gespeichert werden Schlüssel vom Typ `int`.

3.1.1 Hashtabellen

Die abstrakte Klasse `StatCuckooHashSet` enthält den Großteil der Implementierung. Sie lässt aber offen, wie die Hashtabellen realisiert werden. Die Unterklasse `DAryCuckooHashSet` realisiert die Hashtabellen anhand eines zweidimensionalen Integer-Feldes `tables`, wobei `tables[i][j]` den j-ten Behälter der i-ten Hashtabelle darstellt, und die Unterklasse `MultiBucketCuckooHashSet` realisiert die Hashtabellen anhand eines dreidimensionalen Integer-Feldes, wobei `tables[i][j][k]` die k-te Stelle im j-ten Behälter der i-ten Hashtabelle darstellt. Beide Klassen unterstützen die Verwendung von beliebig vielen Hashtabellen, und `MultiBucketCuckooHashSet` ermöglicht das Speichern mehrerer Schlüssel pro Behälter.


<<interface>>
CuckooHashSet
insert(int) : void
delete(int) : void
lookup(int) : boolean
<<interface>>
IntHashFamily
getHashFunction(int) : IntHashFunction
hashFamily
1..1
StatCuckooHashSet
containsZero : boolean
currLoad : int
numTables : int
getStatistics() : Statistics
setStatistics(Statistics) : void
tryToRehash(Collection<Integer>,
List<IntHashFunction>) : boolean
PracticalHashFamily
- upperLimit : int
RandomHashFamily
- upperLimit : int
hashFunctions
2..*
<<interface>>
IntHashFunction
hash(int) : int
statistics
1..1
DAryCuckooHashSet
tables : int[][]
tableSize : int
insert(int,int,int) : void
delete(int,int,int) : void
lookup(int,int,int) : boolean
getNewHashFunctions() : List<IntHashFunction>
<<interface>>
Statistics
reportCollision() : void
reportRehash() : void
reportSuccess() : void
reportTotalFailure() : void
MultiBucketCuckooHashSet
tables : int[][][]
bucketSize : int
CollisionFrequencyDistribution
StatSummary

Da wir bei der theoretischen Untersuchung von einer festen Tabellengröße ausgehen, verzichten wir in unserer Implementierung auf eine dynamische Änderung der Tabellengröße, die in einer praxisorientierten Implementierung durchaus sinnvoll wäre.

3.1.2 Suchen, Löschen und Einfügen

Unsere Implementierung der Methoden `delete` und `lookup` entspricht dem im 1. Kapitel angegebenen Pseudocode.

Die Implementierung der Methode `insert` unterscheidet sich aber an den folgenden zwei Stellen vom Pseudocode:

Symmetrie Im Pseudocode werden Schlüssel immer zuerst in die erste Hashtabelle eingefügt, so dass die erste Hashtabelle im Allgemeinen mehr Elemente enthält als die zweite. Diese Asymmetrie ist zwar vorteilhaft beim Suchen, aber sie führt zu etwas längeren Einfügezeiten. Außerdem fällt die Verteilung der Einfügezeiten nicht mehr monoton ab, weil eine ungerade Anzahl von Schritten wahrscheinlicher ist als eine gerade Anzahl.

Wir haben uns also dafür entschieden, die Schlüssel so einzufügen, dass jede Hashtabelle ungefähr gleich viele Schlüssel enthält. Diese symmetrische Einfügung realisieren wir anhand eines Zählers `currTable`, der angibt, in welche Tabelle eingefügt werden soll. Jedes Mal, dass wir einen Schlüssel an einer Position einer Hashtabelle einfügen (und ggf. dadurch einen anderen Schlüssel hinauswerfen), setzen wir den Zähler auf `currTable` + 1 (modulo der Anzahl der Tabellen).

Absolutes Scheitern Wenn wir den im Pseudocode beschriebenen Algorithmus verwenden würden, könnte eine Einfügung theoretisch zu beliebig vielen Neuaufbauten der Hashtabellen führen. Die Wahrscheinlichkeit einer hohen Anzahl von Neuaufbauten ist zwar bei einem Lastfaktor von 0.25 verschwindend klein, aber in unseren Experimenten handelt es sich oft um höhere Lastfaktoren, bei denen die Hashtabellen öfter neuaufgebaut werden müssen.

In unserer Implementierung definieren wir also eine Konstante `REHASH_LIMIT`, die angibt, wie oft die Hashtabellen neuaufgebaut werden dürfen, bevor wir die aktuelle Einfügung endgültig aufgeben. Wir sagen, dass das Einfügen ***absolut scheitert***, wenn diese maximale Anzahl überschritten wird.

Hilfsmethoden Unsere Implementierung von `insert` verwendet zwei Hilfsmethoden: `tryToInsert` und `insert(int,int,int)`.

Die Methode `tryToInsert` versucht, einen gegebenen Schlüssel einzufügen, und gibt genau dann `true` zurück, wenn das Einfügen ohne Neuaufbau der Hashtabellen gelingt. Das Einfügen zählt als nicht gelungen, wenn die Anzahl der Kollisionen größer als $2n$ ist, wobei n die Größe der bisher eingefügten Schlüsselmenge ist. Die Begründung dafür basiert auf den Beweis zum Satz 1. Wir wissen, dass das Einfügen eines Schlüssels genau

```
public void insert(int elem) throws TotalFailureException {
    if (lookup(elem)) return;

    boolean inserted;
    if (elem == 0) {
        containsZero = true;
        inserted = true;
    } else {
        inserted = tryToInsert(elem);
        if (!inserted) {
            Set<Integer> allElements = toSet();
            allElements.add(elem);
            while (!inserted && currRehashes <= REHASH_LIMIT)
                inserted = tryToRehash(allElements, getNewHashFunctions());
        }
    }

    if (!inserted)
        throw new TotalFailureException(elem);
    else
        currRehashes = 0;
}

private boolean tryToInsert(int elem) {
    int cuckoo = elem; //the element to be inserted
    int egg; //the displaced element

    for (int k = 0; k <= 2 * currLoad; k++) {
        for (int i = 0; i < numTables; i++) {
            int bucket = hashFunctions.get(currTable).hash(cuckoo);
            egg = insert(cuckoo, currTable, bucket);
            if (egg == 0) { //nothing was displaced (success!)
                currLoad++;
                currTable = (currTable + 1) % numTables;
                return true;
            } else { //collision
                cuckoo = egg;
                currTable = (currTable + 1) % numTables;
            }
        }
    }

    currRehashes++;
    return false;
}
```

Abbildung 3.1: Die wesentlichen Zeilen der Implementierung von `insert` und der Hilfsmethode `tryToInsert`. Es wurden z. B. die Zeilen weggelassen, die zum Sammeln von Statistiken dienen.

dann gelingt, wenn der beim Einfügen des Schlüssels traversierte Weg im Kuckucksgraphen höchstens einen Kreis enthält. Enthält der Kuckucksgraph n Kanten, so kann ein solcher Weg niemals länger als $2n$ werden.

Die Methode `insert(int,int,int)` fügt einfach einen Schlüssel an einer gegebenen Stelle ein und gibt ggf. den dadurch hinausgeworfenen Schlüssel zurück.

3.1.3 Hashfunktionen

Da wir bei der theoretischen Untersuchung annehmen, dass die Hashfunktionen zufällige Funktionen sind, verwenden wir auch für unsere Experimente eine Familie pseudozufälliger Funktionen. Jeder Funktion f dieser Familie wird ein Zufallsgenerator zugewiesen. Um für einen Schlüssel x den Hashwert $f(x)$ zu berechnen, geht die Funktion f wie folgt vor: Wenn $f(x)$ vorher noch nie berechnet wurde, wird eine Zufallszahl generiert, zusammen mit x in einer Tabelle gespeichert und zurückgegeben. Soll $f(x)$ wieder berechnet werden, so gibt f einfach den Wert aus der Tabelle zurück.

Solche Hashfunktionen würde man natürlich nie in der Praxis benutzen, unter anderem weil es sinnlos ist, für die Realisierung eines dynamischen Wörterbuchs Hashfunktionen zu verwenden, die selbst voraussetzen, dass eine genügend effiziente Implementierung dynamischer Wörterbücher schon existiert. In unseren Experimenten geht es uns aber eher darum, unsere Implementierung so zu gestalten, dass sie möglichst den Annahmen aus der theoretischen Untersuchung entsprechen.

Für die Experimente, die zur Untersuchung der Leistung von Kuckucks-Hashing in realistischen Situationen dienen, verwenden wir MurmurHash3-Funktionen[1]. Die Implementierung dieser Funktionen entnehmen wir der Guava-Bibliothek.

Die MurmurHash3-Funktionen liefern Hashwerte vom Typ `int`. Um ganze Zahlen aus der Menge $\{0, 1, \ldots, m-1\}$ zu erhalten, wobei m die Größe der Hashtabellen ist, geben wir den Betrag des Hashwertes modulo m zurück.

```
public int hash(int key) {
    int hashValue = func.hashInt(key).asInt();
    return Math.abs(hashValue) % upperLimit;
}
```

3.1.4 Statistik

Zum Sammeln von Statistiken verwenden wir Objekte vom Typ `Statistics`.

Die Unterklasse `CollisionFrequencyDistribution` speichert Paare (k, H_k), wobei H_k die Anzahl der Einfügungen ist, die zu genau k Kollisionen führten.

Die Unterklasse `StatSummary` zählt, wie oft Kollisionen und Neuaufbauten auftreten und wie oft das Einfügen absolut scheitert.

3.2 Experimente

3.2.1 Daten

Für die Experimente, wo wir pseudozufällige Hashfunktionen verwenden, erwarten wir, dass die Ergebnisse unabhängig von den Daten sind. Wir verwenden also die Klasse `SimpleExperiment`, die einfach Zahlen $x \in \mathbb{N}$ mit $x <$ `Integer.MAX_VALUE` der Reihe nach einfügt.

Für die restlichen Experimente benutzen wir die Klasse `IPAddrExperiment`, die IP-Adressen aus einer vom Benutzer angegebenen Datei extrahiert, in vorzeichenbehaftete Zahlen[1] vom Typ `int` umwandelt und in der Reihenfolge einfügt, in der sie in der Datei gefunden wurden.

3.2.2 Beschreibung der Experimente

Einfüge-Experiment Die Experimente, die wir durchführen, basieren auf das folgende Experiment:

Gegeben die Größe n der Schlüsselmenge und die Anzahl N der Einfügungen

Schritt 1 Füge $n - 1$ paarweise verschiedene Elemente ein. Falls das Einfügen dieser Elemente gelingt, speichere die aktuellen Hashfunktionen, damit dieser Zustand später wiederhergestellt werden kann. Breche das Experiment ab, falls das Einfügen auch nach wiederholtem Neuaufbau der Hashtabellen nicht gelingt.

Schritt 2 Führe die folgenden Schritte N-mal aus:

1. Wähle einen Schlüssel x, der noch nicht in der Menge enthalten ist.
2. Füge x ein. Speichere dabei Statistiken, wie die Anzahl der Kollisionen und Neuaufbauten. Falls das Einfügen absolut scheitert, baue die Hashtabellen wieder mit den im 1. Schritt gespeicherten Hashtabellen auf; sonst entferne x, so dass die Größe der Menge wieder $n - 1$ ist.

Dieses Experiment garantiert nicht, dass jeder der N im 2. Schritt gewählten Schlüssel unter genau den gleichen Umständen eingefügt wird. Bei jeder Einfügung werden nämlich möglicherweise Schlüssel verschoben oder sogar die Hashtabellen neu aufgebaut.

Der Lastfaktor bleibt aber immer gleich. Wenn wir stattdessen in jeder Iteration des Experiments alle n Schlüssel einfügen würden, wäre der Lastfaktor für die ersten Elemente wesentlich kleiner, was die Ergebnisse verzerren würde.

[1] Wir wandeln die Adressen in vorzeichenbehaftete Zahlen um, weil Zahlen in Java immer vorzeichenbehaftet sind. Das hat aber keinen Einfluss auf unsere Ergebnisse.

Konfiguration Für jedes Experiment müssen mehrere Parameter gesetzt werden: die Anzahl i der Iterationen, die Anzahl N der Einfügungen pro Iteration, die Größe n der Schlüsselmenge, der Lastfaktor l, die Anzahl d der Hashtabellen, die Größe b der Behälter und ggf. die einzufügenden Daten (falls Daten aus einer Datei eingefügt werden sollen). Diese Parameter werden als Objekte vom Typ `Configuration` gespeichert.

Kollisionen-Experiment

Anzahl der Iterationen:	i	Lastfaktor:	0.25
Anzahl der Einfügungen:	N	Hashtabellen:	2
Größe der Schlüsselmenge:	n	Behältergröße:	1

Das Kollisionen-Experiment führt i Iterationen des Einfüge-Experiments (mit n und N als Eingaben) durch und sammelt dabei Statistiken anhand eines Objekts vom Typ `CollisionFrequencyDistribution`. Die Anzahl der Hashtabellen und die Behältergröße entsprechen den Voraussetzungen unserer Beweise im theoretischen Kapitel. Den Lastfaktor in diesem und im nachfolgenden Experiment hätten wir auf einen beliebigen Wert $l < 0.5$ setzen können. Wir haben uns für $l = 0.25$ entschieden, weil dieser Wert der Voraussetzung im ursprünglichen Kodierungsbeweis von Pătraşcu entspricht.

Zweck dieses Experiments ist, zu sehen, wie schnell die Häufigkeit H_k von Einfügezeiten k mit zunehmendem k abfällt und ob sich die Verteilung der Einfügezeiten so verhält, wie wir es nach Satz 2 erwarten würden.

Neuaufbauten-Experiment

Anzahl der Iterationen:	i	Lastfaktor:	0.25
Anzahl der Einfügungen:	N	Hashtabellen:	2
Größe der Schlüsselmenge:	$n_1, n_2, \ldots$	Behältergröße:	1

Das Neuaufbauten-Experiment führt für jede angegebene Größe n_1, n_2, ... der Schlüsselmenge i Iterationen des Einfüge-Experiments (mit jeweils N Einfügungen) durch und sammelt dabei Statistiken anhand eines `StatSummary`-Objekts. Der Lastfaktor, die Anzahl der Hashtabellen und die Behältergröße entsprechen den Voraussetzungen unserer Beweise im theoretischen Kapitel.

Zweck dieses Experiments ist, zu sehen, wie schnell die durchschnittliche Anzahl der Neuaufbauten pro Einfügung mit zunehmender Größe n der Schlüsselmenge abfällt und ob sich die Verteilung der Neuaufbauten so verhält, wie wir es nach Satz 3 erwarten würden.

Tabellen-Varianten-Experiment

Anzahl der Iterationen:	i	Lastfaktor:	0.01 bis 0.99
Anzahl der Einfügungen:	N	Hashtabellen:	$d_1, d_2, \ldots$
Kapazität der Hashtabellen:	c	Behältergröße:	1

Das Tabellen-Varianten-Experiment führt für jeden Lastfaktor l zwischen 0.01 und 0.99 und für jede angegebene Anzahl d_1, d_2, ... von Hashtabellen i Iterationen des Einfüge-Experiments (mit $n = l \cdot c$ und N als Eingaben) durch. Dabei werden Statistiken anhand eines `StatSummary`-Objekts gesammelt.

Zweck dieses Experiments ist, die Leistung von Kuckucks-Hashing-Varianten, die mehrere Hashtabellen verwenden, zu untersuchen. Uns interessiert insbesondere, wie schnell die durchschnittliche Einfügezeit mit zunehmendem Lastfaktor wächst, und bis zu welchen Lastfaktoren die jeweiligen Varianten mit weniger Gefahr von hohen Einfügezeiten und absolutem Scheitern verwendet werden können.

Behälter-Varianten-Experiment

Anzahl der Iterationen:	i	Lastfaktor:	0.01 bis 0.99
Anzahl der Einfügungen:	N	Hashtabellen:	2
Kapazität der Hashtabellen:	c	Behältergröße:	$b_1, b_2, \ldots$

Das Behälter-Varianten-Experiment führt für jeden Lastfaktor l zwischen 0.01 und 0.99 und für jede angegebene Größe $b_1, b_2, \ldots$ der Behälter i Iterationen des Einfüge-Experiments (mit $n = l \cdot c$ und N als Eingaben) durch. Dabei werden Statistiken anhand eines `StatSummary`-Objekts gesammelt.

Der Zweck dieses Experiments ist analog zu dem des Tabellen-Varianten-Experiments.

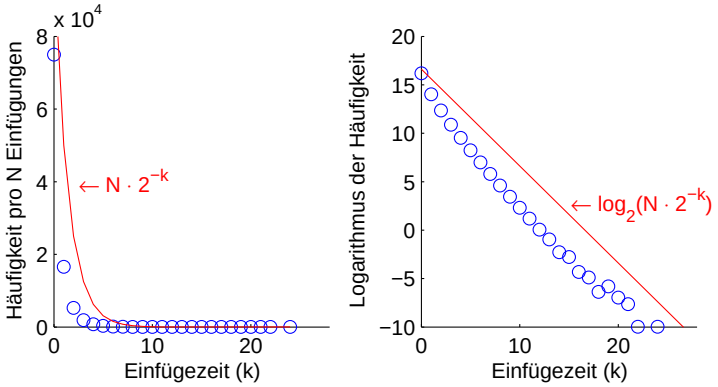
3.3 Ergebnisse

3.3.1 Kollisionen-Experiment

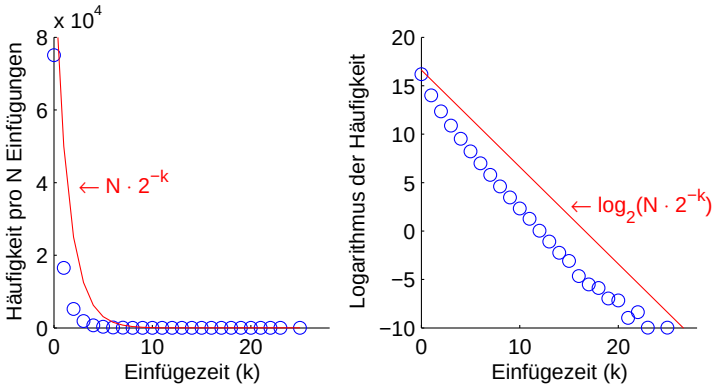
Sei S eine Schlüsselmenge, $x \in S$ und p_k die Wahrscheinlichkeit, dass das Einfügen von x zu $k \in \mathbb{N}$ Kollisionen führt. Nach Satz 2 ist $p_k \leq 2^{-\Omega(k)}$. Bei $N \in \mathbb{N}$ Einfügungen ist also die erwartete Häufigkeit $\mathrm{E}[H_k]$ der Einfügezeit k gegeben durch $\mathrm{E}[H_k] = N \cdot p_k \leq N \cdot 2^{-\Omega(k)}$. Ausführlicher:

$$\exists c > 0\ \exists k_0 > 0\ \forall k > k_0 : \mathrm{E}[H_k] \leq N \cdot 2^{-ck}$$

Das Kollisionen-Experiment bestätigt diese Eigenschaft. In Abbildung 3.2, insbesondere in der logarithmischen Darstellung, ist klar zu sehen, dass die durchschnittliche Häufigkeit der Einfügezeit k schneller abfällt als $N \cdot 2^{-ck}$ mit $c = 1$, egal ob wir pseudozufällige Hashfunktionen oder die effizienteren MurmurHash3-Funktionen verwenden.

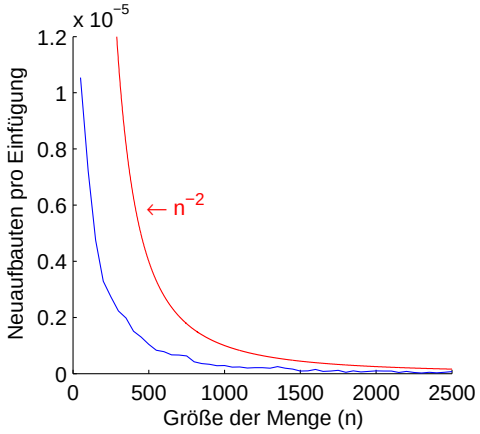


(a) Die Ergebnisse des Experiments, bei dem die Zahlen 0, 1, ..., `Integer.MAX_VALUE` als Daten und pseudozufällige Funktionen zum Berechnen der Hashwerte verwendet wurden.

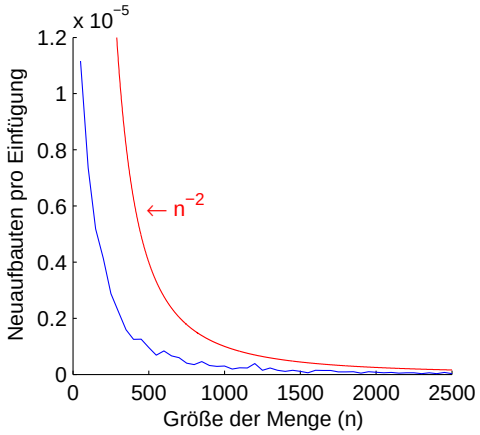


(b) Die Ergebnisse des Experiments, bei dem IP-Adressen als Daten und MurmurHash3-Funktionen zum Berechnen der Hashwerte verwendet wurden.

Abbildung 3.2: Die Ergebnisse des Kollisionen-Experiments (blau) und eine theoretische obere Schranke (rot). Damit das Verhalten ab $k = 5$ besser zu erkennen ist, wird im rechten Diagramm der binäre Logarithmus der Häufigkeiten dargestellt. Das Einfüge-Experiment wurde 1000-mal ausgeführt, mit jeweils $N = 100000$ Einfügungen und Schlüsselmengen der Größe 10000.



(a) Die Ergebnisse des Experiments, bei dem die Zahlen 0, 1, ..., `Integer.MAX_VALUE` als Daten und pseudozufällige Funktionen zum Berechnen der Hashwerte verwendet wurden.



(b) Die Ergebnisse des Experiments, bei dem IP-Adressen als Daten und MurmurHash3-Funktionen zum Berechnen der Hashwerte verwendet wurden.

Abbildung 3.3: Die Ergebnisse des Neuaufbauten-Experiments (blau) und eine theoretische obere Schranke (rot). Das Einfüge-Experiment wurde 10000-mal ausgeführt mit jeweils 10000 Einfügungen.

3.3.2 Neuaufbauten-Experiment

Sei S eine Schlüsselmenge der Größe n, $x \in S$ und p_f die Wahrscheinlichkeit, dass das Einfügen von x scheitert (was einem Neuaufbau der Tabellen entspricht). Nach Satz 3 ist $p_f \leq O(n^{-2})$. Die erwartete Anzahl $\mathrm{E}[B_n]$ der Neuaufbauten pro Einfügung ist also gegeben durch $\mathrm{E}[B_n] = p_f \leq O(n^{-2})$; d. h.

$$\exists c > 0\ \exists k_0 > 0\ \forall k > k_0 : \mathrm{E}[B_n] \leq c \cdot n^{-2}.$$

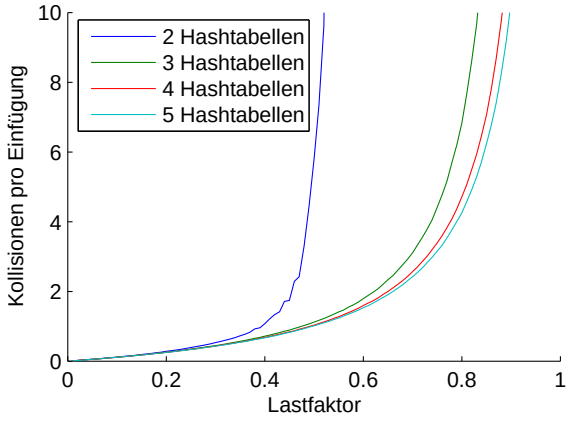
Das Neuaufbauten-Experiment bestätigt diese Eigenschaft. In Abbildung 3.3 fällt die durchschnittliche Anzahl der Neuaufbauten pro Einfügung schneller ab als $c \cdot n^{-2}$ mit $c = 1$.

3.3.3 Varianten-Experimente

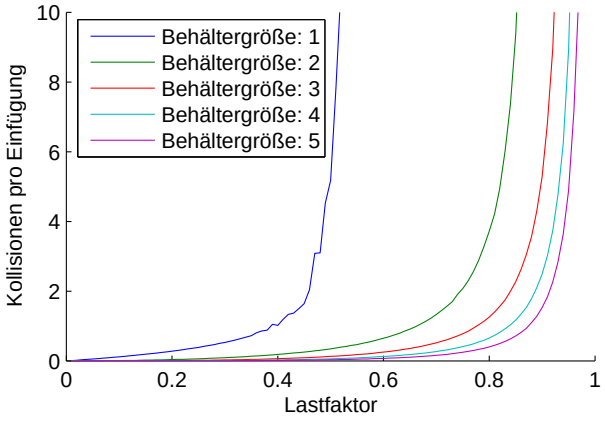
Für die Tabellen- und Behälter-Varianten-Experimente haben wir das Einfüge-Experiment 10-mal ausgeführt mit Kapazität 10000 und jeweils 10000 Einfügungen. Die Ergebnisse beider Experimente zeigen, dass es sich durchaus lohnt, mehr Hashtabellen oder größere Behälter zu verwenden. In Abbildung 3.4 sehen wir, dass die herkömmliche Version mit zwei Hashtabellen und einem Schlüssel pro Behälter schon ab dem Lastfaktor 0.4 unzuverlässig wird. Bei Lastfaktoren größer als 0.5 ist sie unbrauchbar. Die durchschnittliche Anzahl der Kollisionen bei den Varianten, dagegen, bleibt auch bei wesentlich höheren Lastfaktoren wie 0.8 und 0.9 relativ klein.

3.3.4 Realistische Daten und Hashfunktionen

Die Ergebnisse der Experimente, bei denen IP-Adressen als Daten und MurmurHash3-Funktionen zum Berechnen der Hashwerte verwendet wurden, unterscheiden sich nur geringfügig von den Ergebnissen der Experimente, bei denen pseudozufällige Funktionen zum Berechnen der Hashwerte verwendet wurden. Wegen der höheren Effizienz der MurmurHash3-Funktionen konnten die Experimente aber wesentlich schneller durchgeführt werden.



(a) Die Ergebnisse des Tabellen-Varianten-Experiments.



(b) Die Ergebnisse des Behälter-Varianten-Experiments.

Abbildung 3.4: In dieser Abbildung werden nur die Ergebnisse der Experimente angezeigt, wobei die Zahlen 0, 1, ..., `Integer.MAX_VALUE` als Daten und pseudozufällige Funktionen zum Berechnen der Hashwerte verwendet wurden. Wie bei den vorherigen Experimenten sind die Ergebnisse aber fast identisch, wenn wir MurmurHash3-Funktionen und IP-Adressen als Daten verwenden.

Kapitel 4

Schlusswort

Im theoretischen Kapitel haben wir Eigenschaften des Kuckucksgraphen bewiesen, und mittels des Kodierungsbeweises von Mihai Pătraşcu gezeigt, dass das Einfügen eines Schlüssels in amortisiert konstanter erwarteter Laufzeit gelingt und dass das Einfügen einer Schlüsselmenge der Größe n mit Wahrscheinlichkeit $O(1/n)$ zu einem Neuaufbau der Hashtabellen führt.

Der Kodierungsbeweis unterscheidet sich insofern vom ursprünglichen Beweis von Pagh und Rodler, dass wir voraussetzen, dass die Hashfunktionen zufällig aus der Menge aller Funktionen gewählt werden, statt aus einer Hashfamilie, die mit einer hohen Wahrscheinlichkeit $(1, n^\delta)$-universell ist für ein $\delta > 0$. Eine offene Frage wäre also, ob sich der Kodierungsbeweis verallgemeinern lässt, so dass auch eine schwächere Annahme über die Hashfunktionen reicht.

Eine ähnliche Frage wäre, ob sich Kodierungsbeweise für die Varianten mit $d > 2$ Hashtabellen bzw. einer Behältergröße $b > 1$ formulieren lassen.

Im experimentellen Kapitel haben wir eine eigene Implementierung verwendet, um die tatsächliche Leistung des Kuckucks-Hashing-Verfahrens sowohl mit unseren theoretischen Erwartungen als auch mit der Leistung der Tabellen- und Behälter-Varianten zu vergleichen. Die Leistung haben wir anhand der Anzahl der Kollisionen und der Neuaufbauten pro Einfügung gemessen. Es wäre auch interessant gewesen, die Laufzeiten (in Sekunden gemessen) der Varianten mit einander und mit dem herkömmlichen Verfahren zu vergleichen, um zu sehen, wieviel Zeitaufwand wir tatsächlich durch das Verwenden von mehr Hashtabellen bzw. größeren Behältern sparen.

Im experimentellen Kapitel haben wir auch gesehen, dass es effiziente Hashfunktionen (wie die MurmurHash3-Funktionen) gibt, die zu ungefähr so guten Leistungen führen wie zufällige Funktionen. Allgemein ist es aber noch unklar, was für Hashfunktionen unter welchen Bedingungen (inklusive der Art der Daten) gut für das Kuckucks-Hashing-Verfahren geeignet sind.

Bei unserer Implementierung der Tabellen-Variante sind wir immer die Hashtabellen zyklisch durchlaufen. Den einzufügenden Schlüssel haben wir einfach in die aktuelle Hashtabelle eingefügt. Eine andere Möglichkeit, die in der ursprünglichen Arbeit[4] zur Tabellen-Variante beschrieben ist, wäre, eine Breitensuche durchzuführen, um den kürzesten Weg zu einem freien Platz zu finden. Eine praktischere Möglichkeit, die in der selben Arbeit für die Experimente eingesetzt wird, wäre, in jedem Schritt eine Hashtabelle zufällig zu wählen und den aktuellen Schlüssel dort einzufügen. Diese drei Möglichkeiten könnten noch genauer theoretisch untersucht und experimentell mit einander verglichen werden. Insbesondere für die dritte Möglichkeit sind enge Schranken für die Laufzeit gesucht.

Schon 2001, als Kuckucks-Hashing zuerst eingeführt wurde, war es eine vielversprechende Idee. Jetzt, wo unser theoretisches Verständnis des Verfahrens von Jahr zu Jahr wächst, und wo diverse Varianten existieren, die Schwächen des Verfahrens beheben, ist Kuckucks-Hashing zu einem der attraktivsten und spannendsten Hashing-Verfahren geworden.

Literaturverzeichnis

[1] Austin Appleby. MurmurHash3. http://code.google.com/p/smhasher/wiki/MurmurHash3, 3. April 2011.

[2] Yuriy Arbitman, Moni Naor, and Gil Segev. De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming (ICALP '09)*, pages 107–118, 2009.

[3] Martin Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science*, 380(1–2):47–68, 2007.

[4] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. *Theory of Computing Systems*, 38(2):229–248, 2005.

[5] Adam Kirsch and Michael Mitzenmacher. Using a queue to de-amortize cuckoo hashing in hardware. In *Proceedings of the 16th Annual European Symposium on Algorithms*, pages 751–758, 2007.

[6] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Proceedings of the 9th European Symposium on Algorithms (ESA '01)*, 2001.

[7] Mihai Pătraşcu. Cuckoo Hashing. http://infoweekly.blogspot.de/2010/02/cuckoo-hashing.html, 2. Februar 2010. Siehe Anhang A.

Anhang A

Cuckoo Hashing

Es folgt der Blog-Eintrag[7], in dem Mihai Pătraşcu den Kodierungsbeweis zu den Sätzen 2 und 3 skizziert.

Today, we will prove bounds for the basic cuckoo hashing. We are going to place our n keys into two arrays A[1..b] and B[1..b], where $b = 2n$. For this, we will use two hash functions, h and g. When some element x arrives, we first try to put it in A[$h(x)$]. If that location already contains some element y, try to move y to B[$g(y)$]. If that location already contains some z, try to move z to A[$h(z)$], and so on until you find a free spot.

The proper image to have in mind is that of a random bipartite graph. The graph will have b nodes on each side, corresponding to the locations of A and B. In this view, a key x is an edge from the left vertex $h(x)$ to the right vertex $g(x)$.

Simple paths. As a warm-up, let's deal with the case of simple paths: upon inserting x, the update path finds an empty spot without intersecting itself. It turns out that the update time of cuckoo hashing behaves like a geometric random variable:

> The probability that `insert(`x`)` traverses a simple path of length k is $2^{-\Omega(k)}$.

I will prove this by a cute encoding analysis (you know I like encoding proofs). Let's say you want to encode the two hash codes for each of the n keys. As the hash functions h and g are truly random, this requires $H = 2n \log b$ bits on average (the entropy). But what if, whenever some event $\mathcal{E}$ happened, I could encode the hash codes using $H - \Delta$ bits? This would prove that $\Pr[\mathcal{E}] = O(2^{-\Delta})$: there are only $O(2^{H-\Delta})$ bad outcomes that lead to event $\mathcal{E}$, out of 2^H possible ones. Thus, the task of proving a probability upper bound becomes the task of designing an algorithm.

In our case, $\mathcal{E} = \{$`insert(`x`)` traverses a simple path of length k $\}$ and we will achieve a saving of $\Delta = \Omega(k)$. Here is what we put in the encoding:

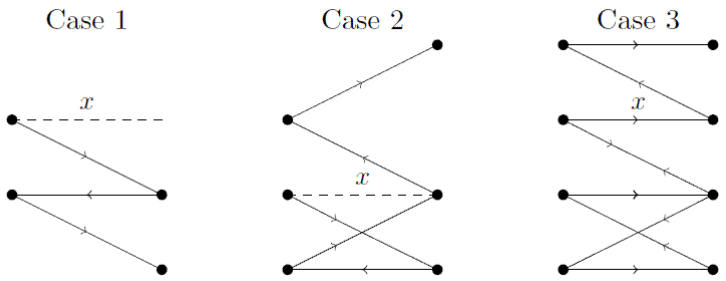
- one bit, saying whether the path grows from $\mathrm{A}[h(x)]$ or $\mathrm{B}[g(x)]$;
- the value k, taking $O(\log k)$ bits;
- all edges of the path, in order, taking $(k-1)\log n$ bits.
- all vertices of the path, in order, taking $(k+1)\log b$ bits.
- the hash codes for all keys not on the path, in order, taking $(n-k)\cdot 2\log b$ bits.

Observe that the hash codes of the k keys on the path are specified using essentially half the information, since a hash code is shared between two edges. Since $\log n = \log(b/2) = \log b - 1$, the encoding saves $\Delta = k - O(\log k)$ bits compared to H.

The intuition for why a k-path occurs with probability $2^{-\Omega(k)}$ is simple. Say I've reached edge y and I'm on the right side. Then, the probability that $\mathrm{B}[g(y)]$ is collision free is at least $1/2$, since there are only n keys mapped to a space of $2n$. In other words, at each point the path stops with probability half. This is exactly what the encoding is saying: we can save one bit per edge, since it takes $\log n$ to encode an edge, but $\log(2n)$ to encode an endpoint.

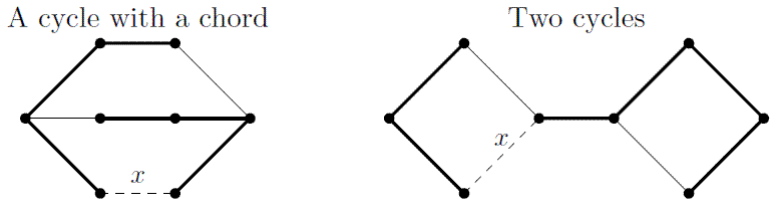
One cycle. Let us now deal with the case that the connected component of x contains one cycle. It is tempting to say that cuckoo hashing fails in this case, but it does not. Here is what might happen to the update path in case a cycle is part of the component (see figure):

1. the path luckily avoids the cycle and finds a free location without intersection itself. Cool.
2. the path reaches $\mathrm{B}[g(x)]$, which is occupied by some key y. Note that this has closed the cycle through the x edge, but the x edge is not actually traversed. Following y to $\mathrm{A}[h(y)]$ must eventually reach a free spot (no more cycles).
3. the path intersects itself. Then, it will start backtracking, flipping elements back to their position before `insert(`x`)`. Eventually, it reaches $\mathrm{A}[h(x)]$, where the algorithm had originally placed x. Following the normal cuckoo rules, x is moved to $\mathrm{B}[g(x)]$ and the exploration from there on must find an empty spot.



As before, we want to show that an update takes time k with probability $2^{-\Omega(k)}$. In cases 1 and 2, we can simply apply the encoding from before, since the path is simple. In case 3, let l be the number of edges until the path meets itself. The number of edges after $\mathrm{B}[g(x)]$ is at least $k - 2l$. Thus, we have a simple path from x of length $\max\{l, k-2l\} = \Omega(k)$, so the old argument applies.

Two cycles. We now arrive at the cases when cuckoo hashing really fails: the bipartite graph contains as a subgraph (1) a cycles with a chord; or (2) two cycles connected by a path (possibly a trivial path, i.e. the cycles simply share a vertex).



From the figure we see that, by removing two edges, we can always turn the bad subgraph into two paths starting at x. We first encode those two paths as above, saving $\Omega(k)$, where $k =$ size of the subgraph. Now we can add to the encoding the two infringing edges. For each, we can specify its identity with $\log n$ bits, and its two end points with $O(\log k)$ bits (a lower order loss compared to the $\Omega(k)$ saving). In return, we know their two hash codes, which are worth $4 \log b$ bits. Thus, the overall saving is at least $2 \log n$ bits.

We have shown that an insertion fails with probability $O(1/n^2)$. By a union bound, cuckoo hashing will handle any fixed set of n elements with probability $1 - O(1/n)$.

This bound is actually tight. Indeed, if three keys x,y,z have $h(x) = h(y) = h(z)$ and $g(x) = g(y) = g(z)$, then cuckoo hashing fails (this is the simplest obstruction subgraph). But such a bad event happens with probability $\binom{n}{3} \cdot b^2/b^6 = \Theta(1/n)$.