Kapitel 12: Hashing (Teil 2)

Perfektes Hashing

Effiziente Algorithmen, SS 2018

Professor Dr. Petra Mutzel

VO 22/23 am 5./10. Juli 2018

Literatur

- Cormen, Leiserson, Rivest, Stein: Algorithmen – Eine Einführung, 4. Auflage, Abschnitt 11.5 Perfektes Hashing (Teile aus Vorlesung und Skript basieren darauf)
- Michael L. Fredman, Janos Komlos und Endre Szemeredi (1984): Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM* 31(3), 538-549 (Statisches Perfektes Hashing)
- Rasmus Pagh und Flemming F. Rodler (2001): Cuckoo hashing. In *Proceedings of the 9th European Symposium on Algorithms (ESA 2001)*, Lecture Notes in Computer Science 2161, Springer, 121-133 (Cuckoo Hashing)
- Rasmus Pagh (2006): Cuckoo Hashing for Undergraduates. Lecture note at IT University of Copenhagen, `www.it-c.dk/people/pagh/papers/cuckoo-undergrad.pdf`

12.5 Performanz von Hashing

im Durchschnitt sehr gut
aber im Worst Case sehr schlecht

Erinnerung Warum ist das so? (hier für offenes Hashing)

Notation

- $B_i = \{x \in S \mid h(x) = i\}$
- $b_i = |B_i|$

im Worst Case $\forall x, y \in S\colon h(x) = h(y)$
$\rightsquigarrow$ Zeit $\Theta(n)$ bei offenem Hashing

im Average Case **Uniformitätsannahme**

Die Hashwerte $h(x)$ ($x \in S$) sind rein zufällig.

klar Uniformitätsannahme gerechtfertigt bei zufälligen Daten

Erwartete Suchzeit bei offenem Hashing

erwartete Zeit für erfolglose Suche bei offenem Hashing

$$= \Theta(1 + \mathsf{E}(b_i))$$

Uniformitätsannahme $\Rightarrow \mathsf{E}(b_i) = \frac{n}{M} = \alpha$

also erwartete Zeit für erfolglose Suche $\Theta(1 + \alpha)$

erwartete Zeit für erfolgreiche Suche bei offenem Hashing

Annahme nach jedem Schlüssel $x \in S$ mit gleicher W'keit suchen

$$\begin{aligned} \text{erw. Zeit} &= \Theta\left(1 + \mathsf{E}\left(\text{Elemente} \in B_{h(x)} \text{ vor } x\right)\right) \\ &= \Theta\left(1 + \mathsf{E}\left(\text{nach } x \text{ eingefügte Elemente} \in B_{h(x)}\right)\right) \end{aligned}$$

Erwartete Zeit erfolgreiche Suche

Definition $X_{i,j} = \begin{cases} 1 & \text{falls } h(x_i) = h(x_j) \\ 0 & \text{sonst} \end{cases}$

Uniformitätsannahme $\Rightarrow \forall i \neq j\colon \text{Prob}\left(X_{i,j} = 1\right) = \frac{1}{M}$

also $\text{E}\left(\text{nach } x \text{ eingefügte Elemente} \in B_{h(x)}\right)$

$$\begin{aligned}
&= \text{E}\left(\sum_{i=1}^{n} \frac{1}{n} \cdot \sum_{j=i+1}^{n} X_{i,j}\right) \\
&= \frac{1}{n} \sum_{i=1}^{n} \sum_{j=i+1}^{n} \text{E}\left(X_{i,j}\right) = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{1}{M} \\
&= \frac{1}{Mn} \sum_{i=1}^{n} (n-i) = \frac{1}{Mn} \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2Mn} = \frac{n-1}{2M} = \Theta(\alpha)
\end{aligned}$$

also erwartete Suchzeit $\Theta(1+\alpha)$

Wappnen gegen den Worst Case

klar mit $n = O(M)$ (also $\alpha = O(1)$) Zeit $\Theta(1)$ $\rightsquigarrow$ sehr anziehend
(vgl. Zeit $\Theta(\log n)$ für AVL-Bäume)

Wunsch Vermeidung des Worst Case

Idee „Verschiebung" des Zufalls

Idee Wähle Hashfunktion h zufällig unabhängig von S.

12.2 Universelle Hashklassen

Annahme $\mathcal{U}$ endlich (also S endlich) und $\mathcal{U} = \{0, 1, \ldots, u-1\}$

triviale Idee Wähle Hashfunktion aus $\{h \colon \mathcal{U} \to \{0, 1, \ldots, M-1\}\}$ zufällig gleichverteilt.

Problem Es gibt $M^{|\mathcal{U}|}$ solche Hashfunktionen $\rightsquigarrow$ nicht praktikabel

Idee Einführung von Hashklassen mit *guten* Eigenschaften

Wozu Klassen von Hashfunktionen?

erwartete Suchzeit für $h \in \mathcal{H}$ zufällig gleichverteilt gewählt?

klar weiterhin $= 1 + \mathrm{E}\left(b_{h(x)}\right) = 1 + \sum_{y \in S \setminus \{x\}} \mathrm{Prob}\,(h(y) = h(x))$

zentraler Unterschied vorhin W'keit über Wahl von y
jetzt W'keit über Wahl von h

Was wünschen wir uns?

Definition 12.1

Eine Hashklasse $\mathcal{H} \subseteq \{h \colon \mathcal{U} \to \{0, 1, \ldots, M-1\}\}$ heißt c-universell, wenn bei zufälliger Wahl von $h \in \mathcal{H}$ für beliebige Schlüssel $x \neq y \in \mathcal{U}$ gilt:

$$\mathrm{Prob}\,(h(x) = h(y)) \leq \frac{c}{M}$$

Suchzeiten bei c-universellen Hashklassen

Theorem 12.4

Die erwartete Suchzeit bei zufälliger Wahl von h aus einer c-universellen Hashklasse $\mathcal{H}$ beträgt bei erfolgreicher und erfolgloser Suche $\Theta(1 + c \cdot \alpha)$.

Beweis.

Rechnung wie im Uniformfall mit W'keit c/M statt $1/M$
$\rightsquigarrow \Theta(1 + c \cdot \alpha)$ □

Gibt es überhaupt c-universelle Hashklassen mit kleinem c?

Die Hashklasse $\mathcal{H}_l$

Definition 12.2: Hashklasse $\mathcal{H}_l$

Für $p \geq |\mathcal{U}|$ prim ist $\mathcal{H}_l = \{h_{a,b} \mid 0 < a < p, 0 \leq b < p\}$ eine Klasse von Hashfunktionen mit $h_{a,b}(x) := ((a \cdot x + b) \bmod p) \bmod M$.

Theorem 12.3

$\mathcal{H}_l$ ist 1-universell.

Beweis des Theorems

Seien $x \neq y \in \mathcal{U} = \{0, 1, \ldots, u-1\}$ beliebig.
Seien $a \in \{1, \ldots, p-1\}$, $b \in \{0, \ldots, p-1\}$ rein zufällig.
Beobachtung p prim $\rightsquigarrow \{1, \ldots, p-1\} = \mathbb{Z}_p^*$
also $x' := (ax+b) \bmod p$ und $y' := (ay+b) \bmod p$ verschieden

also $h(x) = h(y) \Leftrightarrow x' \bmod M \equiv y' \bmod M$ mit $x' \neq y'$

klar $\forall i \in \mathcal{U}\colon |\{j \in \mathcal{U} \mid i \equiv j \bmod M\}| \leq \left\lceil \frac{|\mathcal{U}|}{M} \right\rceil$

also Prob $(h(x) = h(y)) \leq \frac{\lceil u/M \rceil - 1}{u-1} = \frac{\lceil (u-M)/M \rceil}{u-1}$

$$\leq \frac{(u-M+(M-1))/M}{u-1} \leq \frac{1}{M}$$

Zur Implementierung

Einwand alle Überlegungen setzen voraus, dass $|S| = n$ vorab bekannt

Idee dynamische Anpassung der Hashtabellengröße

- Fixiere vorab gewünschten Lastfaktor α'.
- Führe Buch über aktuellen Lastfaktor α.
- Wenn $\alpha > \alpha'$ gilt,
  - Erzeuge neue Hashtabelle doppelter Größe.
  - Wähle zufällig neue Hashfunktion.
  - Trage alle Einträge aus alter Tabelle in neue Tabelle gemäß neuer Hashfunktion. (Bemerkung vorheriges Search unnötig)
  - Lösche alte Hashtabelle.

Beobachtungen bei $\leq n$ Einträgen zu jeder Zeit...

- Gesamtplatzbedarf $O(n)$
- Gesamtzeitbedarf fürs Umkopieren $O(n)$
- erwarteter Gesamtzeitbedarf für k Operationen $O(k)$

12.3 Statisches Perfektes Hashing

bis jetzt alle Aussagen „im Erwartungswert“

Kann man bessere Garantien für die Performanz bekommen?

Perfektes Hashing: Suche dauert im Worst-Case $O(1)$

Anmerkung Worst-Case-Zeit $O(1)$ je Operation verschieden von amortisierter Worst-Case-Zeit $O(1)$ je Operation

Statisches Hashing Schlüssel werden genau einmal eingefügt und dann ist nur Suche erlaubt (z. B. CD-ROM, Wörter in Programmiersprache)

Statisches Perfektes Hashing Aufbau in erwarteter Polynomialzeit und Suchoperation in Worst-Case-Zeit $O(1)$ für statisches Wörterbuch

Auf dem Weg zum statisch perfekten Hashing

Wie erreichen wir Worst-Case-Zeit $O(1)$ je Search?

triviale Beobachtung — wenn unter h kollisionsfrei, dann sicher Zeit $O(1)$

Wie erreichen wir Kollisionsfreiheit?

triviale Beobachtung — Wähle M groß genug.

Lemma 12.5

Beim Hashen von n Schlüsseln in eine Hashtabelle der Größe $M := n^2$ mit einer uniform zufällig gewählten Hashfunktion $h \in \mathcal{H}_l$ gibt es mit Wahrscheinlichkeit $\geq 1/2$ keine Kollision.

Beweis von Lemma 12.5

klar Kollision $\Leftrightarrow h(x) = h(y)$ für $x \neq y \in S$

klar Anzahl solcher Paare $= \binom{n}{2}$

also für jedes Paar Kollisionsw'keit $1/M$ (Theorem 12.3)

also erw. Anzahl Kollisionen $\mathsf{E}\,(K) = \binom{n}{2} \cdot \frac{1}{M}$
$= \binom{n}{2} \cdot \frac{1}{n^2} = \frac{n(n-1)}{2n^2} < \frac{1}{2}$

klar $K \geq 0$ also Markow-Ungleichung anwendbar

damit $\mathsf{Prob}\,(K \geq 1) < 1/2$

also $\mathsf{Prob}\,(\text{keine Kollision}) = 1 - \mathsf{Prob}\,(K \geq 1)\, 1/2$ $\square$

Lemma 12.5 kritisch hinterfragt

klar quadratischer Platz völlig inakzeptabel

Geht es nicht auch mit $M \ll n^2$?

Beobachtung $\text{Prob}\,(\text{keine Kollision}) = \left(1 - \frac{1}{M}\right)^{\binom{n}{2}}$
$= \left(1 - \frac{1}{M}\right)^{M \cdot \binom{n}{2}/M} \leq e^{-\binom{n}{2}/M} = e^{-\Theta(n^2/M)}$

Fazit Hashtabellengrößen $M = o(n^2/\log n)$ chancenlos

also bessere Idee benötigt

Idee für statisches perfektes Hashing

Beobachtung Hashtabellengröße $M' \rightsquigarrow k \cdot M'$
$\widehat{=}$ Platz k je ursprünglichem Bucket
wenn k groß genug, dann Suche in O(1)

Idee Bucketgröße individuell und adaptiv bestimmen

Beobachtungen Wir kennen die Schlüsselmenge (statisch)
Wir können alle b_i vorher berechnen

Idee Zweistufige Hashtabelle:
erste Stufe: $M = n$, zweite Stufe für Buckets

Erinnerung Bucketgröße b_i^2 ausreichend (Lemma 12.5)

Idee für statisches perfektes Hashing

Erinnerung Bucketgröße b_i^2 ausreichend (Lemma 12.5)

Aber ist $\sum_{i=0}^{M-1} b_i^2$ nicht auch schon quadratisch?

Lemma 12.6

Sei $M = n$, $h \in \mathcal{H}_l$ uniform zufällig gewählt,
$b_i := |\{x \in S \mid h(x) = i\}|$.

$$\mathsf{E}\left(\sum_{i=0}^{M-1} b_i^2\right) < 2n$$

Beweis von Lemma 12.6

klar $\sum_{i=0}^{M-1} b_i = n$

klar $a^2 = \frac{2a(a-1)}{2} + a = 2\binom{a}{2} + a$

also
$$\mathsf{E}\left(\sum_{i=0}^{M-1} b_i^2\right) = \mathsf{E}\left(\sum_{i=0}^{M-1} b_i\right) + 2\sum_{i=0}^{M-1} \mathsf{E}\left(\binom{b_i}{2}\right)$$
$$= n + 2\sum_{i=0}^{M-1} \mathsf{E}\left(\binom{b_i}{2}\right)$$

Beobachtung $\sum_{i=0}^{M-1} \binom{b_i}{2} = |\{x \neq y \in S \mid h(x) = h(y)\}|$

Definiere $X_{x,y} := \begin{cases} 1 & \text{falls } h(x) = h(y) \\ 0 & \text{sonst} \end{cases}$

damit $\sum_{i=0}^{M-1} \binom{b_i}{2} = \sum_{x \neq y \in S} X_{x,y}$

Beweis von Lemma 12.6 (Fortsetzung)

Wir haben $\mathsf{E}\left(\sum_{i=0}^{M-1} b_i^2\right) = n + 2\sum_{i=0}^{M-1} \mathsf{E}\left(\binom{b_i}{2}\right)$

mit $\sum_{i=0}^{M-1} \binom{b_i}{2} = \sum_{x \neq y \in S} X_{x,y}$

Erinnerung $\mathsf{E}\left(X_{x,y}\right) = 1/M$ (Theorem 12.3)

also $\mathsf{E}\left(\sum_{i=0}^{M-1} b_i^2\right) = n + 2 \cdot \binom{n}{2} \cdot \frac{1}{M}$

$= n + \frac{n(n-1)}{n} < 2n$

□

Algorithmus 12.7

1. Repeat
2. Wähle $h \in \mathcal{H}_l$ uniform zufällig.
3. Für alle $i \in \{0, 1, \ldots, M-1\}$ $b_i := 0$; $B_i := \emptyset$
4. Für alle $x \in S$ $b_{h(x)} := b_{h(x)} + 1$; $B_{h(x)} := B_{h(x)} \cup \{x\}$
5. Until $\sum_{i=0}^{M-1} b_i^2 < 4n$.
6. Für alle $i \in \{0, 1, \ldots, M-1\}$
7. Repeat
8. gut := true
9. Initialisiere leere Hashtabelle H_i der Größe b_i^2.
10. Wähle $h_i \in \mathcal{H}_l$ uniform zufällig.
11. Für alle $x \in B_i$
    If $H_i[h_i(x)]$ belegt Then gut := false
    Else Speichere x in $H_i[h_i(x)]$.
12. Until gut

Über statisch perfektes Hashing

Theorem 12.8

Algorithmus 12.7 konstruiert in erwarteter Zeit $O(n)$ eine Datenstruktur der Größe $O(n)$ mit zusätzlichem Speicherbedarf $O(n)$ und wählt dabei Hashfunktionen so aus, dass jede anschließende Search-Operation nur konstante Zeit braucht.

Beweis.

klar Search(x) mit Berechnung von $h(x)$ und $h_{h(x)}$ ✓

klar Speicherplatz explizit so beschränkt ✓

offen erwartete Laufzeit

Erwartete Laufzeit der ersten Phase

Erste Phase (Zeilen 1–5)

Erinnerung $\mathsf{E}\left(\sum_{i=0}^{M-1} b_i^2\right) < 2n$ (Lemma 12.6)

klar $\sum_{i=0}^{M-1} b_i^2 \geq 0$

also $\text{Prob}\left(\sum_{i=0}^{M-1} b_i^2 \geq 4n\right) < 2n/4n = 1/2$ (Markow)

also im Erwartungswert ≤ 2 Durchläufe

also erwartete Laufzeit $O(n)$

Erwartete Laufzeit der zweiten Phase

Zweite Phase (Zeilen 6–12)

Betrachte einen Repeat-Until-Schleifendurchgang

Erinnerung W'keit für Ende $\geq 1/2$ (Lemma 12.5)

also erwartete Laufzeit $O\left(\sum_{i=0}^{M-1} b_i^2\right)$

Erinnerung Vorbedingung $\sum_{i=0}^{M-1} b_i^2 < 4n$

also Gesamtlaufzeit $O(n)$ $\square$

Fazit statisches perfektes Hashing praktikabel
und leicht zu implementieren

Erweiterung auf Dynamisches Perfektes Hashing

Dynamisches Perfektes H. Worst-Case-Zeit $O(1)$ je Search und amortisierte erwartete Zeit $O(1)$ für Insert und Delete

Zentrale Ideen

- benutze gleichen Ansatz wie für statisches perfektes Hashing
- benutze Verdopplungsstrategie für Größe der Hashtabelle
- nach n Updates oder bei ungünstiger Verteilung, kompletter Neuaufbau der Hashtabelle
- offen Werte passend konkret definieren und nachrechnen, dass alles passt $\rightarrow$ aber sehr komplex

Details Dietzfelbinger, Karlin, Mehlhorn, Meyer auf der Heide, Rohnert, Tarjan (1994): Dynamic perfect hashing: upper and lower bounds, SIAM J. Comput. 23, 738-761.

12.4 Cuckoo Hashing

einfach und praktisch effizient

Ideen

- Benutze zwei Hashtabellen T_1 und T_2 der Größe jeweils $M = (1 + \varepsilon)n$. (ε konstant)
- Falls n nicht bekannt, benutze übliche Verdopplungstaktik.
- Benutze zwei Hashfunktionen h_1 und h_2, für jede Hashtabelle eine.
- Speichere x entweder in $T_1[h_1(x)]$ oder in $T_2[h_2(x)]$:
- Falls Platz $T_1[h_1(x)]$ besetzt ist, dann drängt der zu speichernde Schlüssel x den Konkurrenten y heraus. Dieser drängt nun seinerseits seinen Konkurrenten aus seinem alternativen Platz $T_2[h_2(y)]$, u.s.w. (deswegen Kuckucks-Hashing: wird aus seinem Nest gestoßen)
- Führe komplettes Rehashing durch, wenn es Probleme gibt.

klar Search(x) einfach

x vorhanden $\Leftrightarrow$ x steht in $T_1[h_1(x)]$ oder in $T_2[h_2(x)]$

klar Delete = Search + Entfernen

Cuckoo Hashing: Insert

Eingabe x

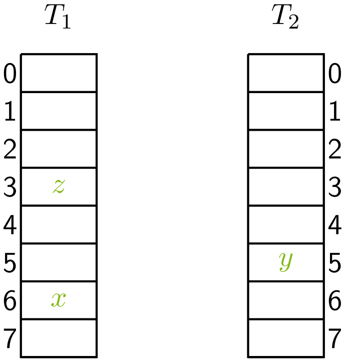
1. Search(x). Falls vorhanden, STOP.
2. Für $k \in \{1, 2, \ldots, k_{\max}\}$
3. Vertausche x und den Inhalt von $T_1[h_1(x)]$.
4. Falls x leer, STOP.
5. Vertausche x und den Inhalt von $T_2[h_2(x)]$.
6. Falls x leer, STOP.
7. RehashAll
8. Insert(x)

Wahl des Parameters $k_{\max}$ später

nach M^2 Insert-Aufrufen (inkl. Versuche Z.8) auf jeden Fall Aufruf von RehashAll

RehashAll wählt h_1 und h_2 randomisiert neu,
fügt alle Elemente neu ein

Beispiel Insert (erfolgreich)

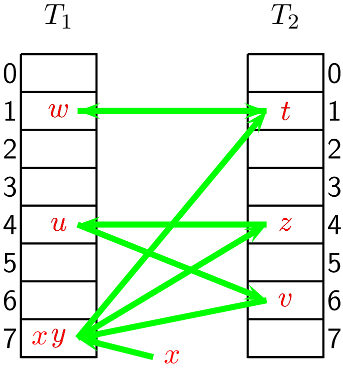


$h_1(y) = 6,\ h_2(y) = 5$
$h_1(z) = 3,\ h_2(z) = 5$

Insert(x)

$h_1(x) = 6,\ h_2(x) = 7$

Beispiel Insert (erfolglos)



$h_1(t) = 1,\ h_2(t) = 1$
$h_1(u) = 4,\ h_2(u) = 6$
$h_1(v) = 7,\ h_2(v) = 6$
$h_1(w) = 1,\ h_2(w) = 1$
$h_1(y) = 7,\ h_2(y) = 4$
$h_1(z) = 4,\ h_2(z) = 4$

Insert(x)

$h_1(x) = 7,\ h_2(x) = 1$

Analyseresultate

Lemma 12.11

Cuckoo-Hashing benötigt Worst-Case Zeit $O(1)$ für die Operationen Search und Delete sowie eine erwartete amortisierte Rechenzeit $O(1)$ für die Operation Insert.

Die Original-Analyse ist relativ komplex und erfordert Konzepte, die im Rahmen einer Bachelor-Vorlesung nicht geeignet ist.

Wir analysieren eine Variante von Cuckoo-Hashing, die nur eine Hashtabelle benötigt. Die Analyse ist deutlich einfacher und für Vorlesungen geeignet. (Achtung: nicht im Skript, sondern s.o. Literatur)

Cuckoo Hashing Variante von Pagh

Ideen

- Benutze eine Hashtabelle T der Größe $M \geq 2cn$ mit $c > 1$ konstant.
- Falls n nicht bekannt, benutze übliche Verdopplungstaktik.
- Benutze zwei Hashfunktionen h_1 und h_2
- Speichere x entweder in $T[h_1(x)]$ oder in $T[h_2(x)]$:
- Falls Platz $T[h_1(x)]$ besetzt ist, dann drängt der zu speichernde Schlüssel x den Konkurrenten y heraus. Dieser drängt nun seinerseits seinen Konkurrenten aus seinem alternativen Platz $T[h_2(x)]$ oder $T[h_1(x)]$, u.s.w. (deswegen Kuckucks-Hashing: wird aus seinem Nest gestoßen)
- Führe komplettes Rehashing durch, wenn es Probleme gibt.

Cuckoo Hashing Variante: Insert

Eingabe x

1. Search(x). Falls vorhanden, STOP.
2. pos $= h_1(x)$
3. Für $k \in \{1, 2, \ldots, n\}$ // max. n Versuche
4. Vertausche x und den Inhalt von $T[pos]$.
5. Falls x leer, STOP.
6. Falls pos $== h_1(x)$ dann pos$= h_2(x)$ sonst pos$= h_1(x)$
7. RehashAll
8. Insert(x)

RehashAll wählt h_1 und h_2 randomisiert neu,
fügt alle Elemente neu ein

Annahmen für die Analyse

- Die verwendeten Hashfunktionen h_1 und h_2 besitzen die folgende Eigenschaft: jeder Funktionswert $h_i(x)$ nimmt mit Wahrscheinlichkeit $1/M$ einen festen Wert in $\{0, 1, \ldots, M-1\}$ an. (stärker als 1-universell)
- Die angenommenen Funktionswerte sind unabhängig voneinander.
- Die Hashfunktionswerte können in konstanter Zeit berechnet werden.

Theorem 12.12

Die Cuckoo-Hashing Variante benötigt Worst-Case Zeit $O(1)$ für die Operationen Search und Delete sowie für eine Folge von $\epsilon n < n$ Operationen eine erwartete amortisierte Rechenzeit $O(1)$ für die Operation Insert ($\epsilon < 1$ konstant).

klar Search(x) und Delete(x) einfach

x vorhanden $\Leftrightarrow$ x steht in $T[h_1(x)]$ oder in $T[h_2(x)]$

zu zeigen Zeit für Insert

Der Kuckucksgraph

Definiere Kuckucks-Graph: Knoten sind Positionen in Tabelle, Kanten entsprechen der beiden alternativen Positionen für die enthaltenen Elemente

Beobachtung Insert(x) kann nur Elemente besuchen, die durch Pfad in G mit x verbunden sind

Definiere den Bucket von x: Alle Positionen in der Hashtabelle, die im Kuckucksgraphen mit x durch einen Weg verbunden sind.

Zunächst: Analyse für ein Insert ohne Rehash

Lemma 12.13

Für je zwei verschiedene Elemente x und y ist die Wahrscheinlichkeit, dass x und y im gleichen Bucket sind, durch $O(1/M)$ beschränkt.

Der Beweis zu Lemma 12.13 (inklusive Lemma 12.16) wurde in der VO aus Zeitgründen nicht mehr gemacht, er ist also nicht prüfungsrelevant.

Lemma 12.14

Die Zeit für eine Operation Insert(x) auf einem Element x ist durch $O($ Anzahl der Elemente im Bucket von $x)$ beschränkt.

Lemma 12.15

Die erwartete Anzahl der Elemente in dem Bucket von x ist durch $O(1)$ beschränkt.

Beweise gleich

Aus Lemma 12.14 und Lemma 12.15 ergibt sich direkt der Beweis zu Theorem 12.12 (ohne Rehash).

Wir zeigen später, dass die Anzahl der erwarteten Rehash-Ausführungen durch 2 beschränkt ist.

Beweis von Lemma 12.15

Lemma 12.15

Die erwartete Anzahl der Elemente in dem Bucket von x ist durch $O(1)$ beschränkt.

Beweis:

- Wir analysieren Insert(x), wobei x nicht in T ist
- Sei S die Menge der in T enthaltenen Elemente
- Laut Lemma 12.13 ist die Wahrscheinlichkeit, dass die Operation ein festes Element $y \in S$ besucht, in $O(1/M)$.
- Damit ist die erwartete Zeit für ein festes Element $y \in S$ in $O(1/M)$.
- Für alle Elemente in S ist die erwartete Zeit in $O(|S|/M) = O(1)$, da $M \geq n \geq |S|$.

Beweis von Lemma 12.14

Lemma 12.14

Die Zeit für eine Operation auf einem Element x ist durch $O(\text{Anzahl der Elemente im Bucket von } x)$ beschränkt.

Beweis:

- Insert kann nur dann n Austausche nach sich ziehen, wenn es in einen Kreis im Kuckucksgraphen gerät.
- Aber dann wird auch ein Rehash aufgerufen. Dieser Fall wird hier nicht betrachtet.
- Also wird jedes Element nur konstant oft betrachtet.

Auf dem Weg zum Beweis zu Lemma 12.13

Lemma 12.16

Sei $M \geq 2cn$ für eine Konstante $c > 1$. Für je zwei Positionen i und j, ist die Wahrscheinlichkeit, dass in dem Kuckucksgraphen ein Pfad von i nach j der Länge $l \geq 1$ existiert, der ein kürzester (i,j)-Pfad ist, höchstens c^{-l}/M.

Beweis:

- Induktion über l. Sei $l = 1$.
- Die Wahrscheinlichkeit für ein Element genau die Positionen i und j als alternative Plätze zugewiesen zu bekommen, ist $\leq (1/M)(1/M) + (1/M)(1/M) = 2/M^2$.
- Für alle Elemente in S ist diese Wahrscheinlichkeit begrenzt durch $\sum_{x \in S} 2/M^2 \leq 2n/M^2 \leq c^{-1}/M$.

- Im Induktionsschritt betrachten wir die Wahrscheinlichkeit, dass ein Pfad der Länge $l > 1$ existiert, aber kein Pfad der Länge $< l$.
- Dies ist nur der Fall, wenn für eine Position k gilt:
- Es existiert ein kürzester (i, k)-Weg der Länge $l - 1$, der nicht durch j geht und es gibt eine Kante von k nach j
- Nach I.V. ist die Wahrscheinlichkeit für ersteres durch c^{-l+1}/M beschränkt und die Wahrscheinlichkeit für letzteres durch $2/M^2 \leq c^{-1}/M$.
- Zusammengenommen ist die Wahrscheinlichkeit für ein spezielles k kleiner gleich c^{-l}/M^2.
- Summiert über alle M Positionen von k, erhalten wir die Wahrscheinlichkeit für einen Weg der Länge l ist höchstens c^{-l}/M.

Beweis von Lemma 12.13

Lemma 12.13

Für je zwei verschiedene Elemente x und y ist die Wahrscheinlichkeit, dass x und y im gleichen Bucket sind, durch $O(1/M)$ beschränkt.

- Wenn x und y dem gleichen Bucket angehören, dann existiert ein Pfad der Länge l zwischen den Mengen $\{h_1(x), h_2(x)\}$ und $\{h_1(y), h_2(y)\}$.
- Aus Lemma 12.16 folgt, dass die Wahrscheinlichkeit hierfür durch folgenden Term begrenzt ist $4\sum_{l=1}^{+\infty} c^{-l}/M = \frac{4}{c-1}/M = O(1/M)$.

Rehashing

Lemma 12.17

Die erwarteten amortisierten Kosten eines Rehashing-Aufrufs bei einem Insert sind in $O(1/n)$ für jede Konstante $c > 1$.

Wir beweisen hier aus Vereinfachungsgründen etwas leicht schwächeres:

Lemma 12.18

Die erwarteten amortisierten Kosten eines Rehashing-Aufrufs bei einem Insert sind in $O(1)$ für jede Konstante $c \geq 3$.

Beweis von Lemma 12.18

Lemma 12.18

Die erwarteten amortisierten Kosten eines Rehashing-Aufrufs bei einem Insert sind in $O(1)$ für jede Konstante $c \geq 3$.

- Annahme: Rehash fügt ϵn Elemente ein, wobei $\epsilon < 1$
- Sei S' die Menge der Elemente, die zum Zeitpunkt des Einfügens eines Elements in T enthalten ist.
- Es kann nur dann einen Rehash-Aufruf geben, wenn im dazugehörigen Kuckucksgraphen ein Kreis enthalten ist.
- Aus Lemma 12.16 folgt: die Wahrscheinlichkeit, dass in G ein kürzester Pfad von i nach i der Länge l enthalten ist, ist höchstens c^{-l}/M.
- Summiert über alle möglichen Knoten i: $Mc^{-l}/M = c^{-l}$.
- Summiert über alle möglichen Längen l: $\sum_{l=1}^{+\infty} c^{-l} = \frac{1}{c-1}$.

Beweis von Lemma 12.18 ff

- Wir haben: Die Wahrscheinlichkeit, dass in G ein Kreis enthalten ist, ist höchstens $\frac{1}{c-1} \leq \frac{1}{2}$.
- Die Wahrscheinlichkeit, dass zwei Rehash-Aufrufe notwendig sind, ist höchstens $\frac{1}{2}\frac{1}{2} = \frac{1}{4}$, u.s.w.
- Die erwartete Anzahl an Rehash-Aufrufen ist also höchstens $\sum_{i=1}^{+\infty} i2^{-i} = 2$ (geom. Reihe).
- Die Laufzeit eines vollständigen Rehash ist in $O(n)$ (nach n Versuchen Abbruch und Wiederholung, aber das nicht zu oft).
- Dann ist die erwartete Zeit für alle Rehash-Operationen $O(n/n\epsilon) = O(1/\epsilon)$ per Einfügung.
- Also sind dann die erwarteten amortisierten Kosten von Rehashing konstant.

Bemerkungen zu Cuckoo Hashing

Experimentelle Resultate von Pagh und Rodler (2003) zeigen:

- Cuckoo Hashing ist sehr sensitiv gegenüber der verwendeten Hashfunktion: Z.B. führte die multiplikative Hashfunktion zu schlechten Resultaten. Die folgenden Resultate wurden für geeignete Hashfunktion erzielt.
- Search: Lineare Sondierung war das beste Verfahren, dann folgte Cuckoo-Hashing, und danach Hashing mittels verketteter Listen (wegen Cache-Effekten)
- Insert: Lineare Sondierung war das beste Verfahren, dann folgten verkettete Listen, und danach kam Cuckoo-Hashing (1.2 bis 2 Mal so lange wie Linear Probing).
- Die Autoren verweisen darauf, dass Lineare Sondierung das beste Verfahren war, dann folgten verkettete Listen, und danach kam Cuckoo-Hashing (1.2 bis 2 Mal so lange wie Linear Probing).

Bemerkungen zu Cuckoo Hashing

Dietzfelbinger und Schellbach (SODA 2009):

- Theoretische Analyse zeigt, dass bei Verwendung der Multiplikativen Klasse der Hashfunktionen die Fehlerquote von Cuckoo Hashing hoch ist, selbst wenn die Größe der Hashtabellen auf $M = 4n$ vergrößert wird.
- Dasselbe gilt bei Verwendung der Hashklasse $\mathcal{H}_l$
- Experimentelle Resultate: quadratische Hashklassen funktionieren sehr gut.