

SS 2021

Algorithmen und Datenstrukturen

5. Kapitel

Hashverfahren

Martin Dietzfelbinger

Mai 2021

5.1 Grundbegriffe

Gegeben: Universum U (auch *ohne Ordnung*) und Wertebereich R .

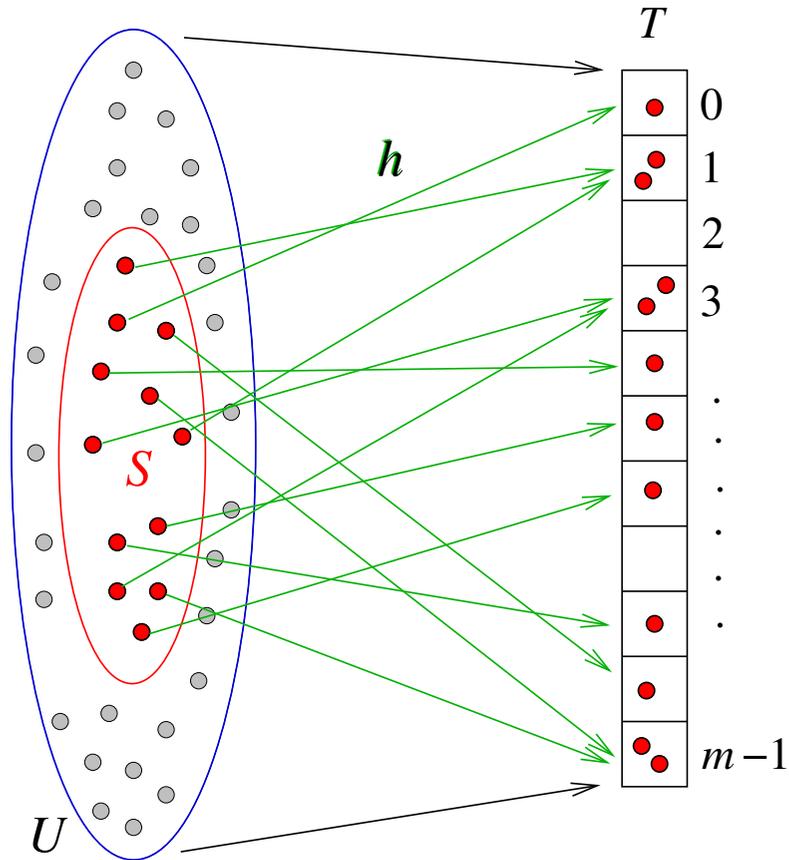
Hashverfahren (oder **Schlüsseltransformationsverfahren**) implementieren ein **Wörterbuch** (alias **dynamische Abbildung**, **assoziatives Array**)

$$f: S \rightarrow R, \text{ mit } S \subseteq U \text{ endlich.}$$

Ziel:

Zeit $O(1)$ pro Operation
„im Durchschnitt“
unabhängig vom Umfang der Datenstruktur!

Gegensatz **Suchbäume:** Zeit $O(\log(n))$ pro Operation.



Hashfunktion* $h: U \rightarrow [m]$

U : Universum der Schlüssel

$[m] = \{0, \dots, m - 1\}$:
Indexbereich für Tabelle T

$S \subseteq U$, $|S| = n$

Definitionsbereich von f .

* to hash (*engl.*): kleinhacken, kleinschneiden

Grundansatz:

Speicherung von $f: S \rightarrow R$ in Tabelle $T[0..m-1]$.

Aus Schlüssel $x \in U$ wird ein Index $h(x) \in [m]$ **berechnet**.

„ x wird in $h(x)$ umgewandelt/**transformiert**.“

Idealvorstellung:

Speichere Paar $(x, f(x))$ in Zelle $T[h(x)]$.

Beispiel: $U = \{A, \dots, Z, a, \dots, z\}^{3..20}$

Nichtleere Wörter aus lateinischen Buchstaben mit mindestens drei und höchstens 20 Buchstaben.

$h: U \rightarrow [13]$ definiert durch

$$h(c_1c_2c_3 \cdots c_r) = \text{num}(c_3) \bmod 13,$$

wobei

$$\begin{array}{rclcl} \text{num}(A) & = & \text{num}(a) & = & 0 \\ \text{num}(B) & = & \text{num}(b) & = & 1 \\ & & \vdots & & \vdots \\ \text{num}(Z) & = & \text{num}(z) & = & 25 \end{array}$$

* $x \bmod m$: Rest bei der Division von x durch m .

In Tabelle:

x	$f(x)$	$\text{num}(c_3)$	$h(x)$
Januar	31	13	0
Februar	28	1	1
März	31	4	4
April	30	17	4
Mai	31	8	8
Juni	30	13	0
Juli	31	11	11
August	31	6	6
September	30	15	2
Oktober	31	19	6
November	30	21	8
Dezember	31	25	12

$$h(c_1 \dots c_r) = \text{num}(c_3) \bmod 13$$

j	$(x, f(x))$ mit $h(x) = j$
0	(Januar, 31); (Juni, 30)
1	(Februar, 28)
2	(September, 30)
3	
4	(Maerz, 31); (April, 30)
5	
6	(August, 31); (Oktober, 31)
7	
8	(Mai, 31); (November, 30)
9	
10	
11	(Juli, 31)
12	(Dezember, 31)

Kollisionen bei $j = 0, 4, 6, 8$.

Definition 5.1.1

Eine Funktion $h: U \rightarrow [m]$ heißt eine **Hashfunktion**.

Wenn $S \subseteq U$ gegeben ist (oder ein $f: S \rightarrow R$), verteilt h die Schlüssel in S auf $[m]$.

$B_j := \{x \in S \mid h(x) = j\}$ heißt **Behälter** („bucket“).

Idealfall: Die Einschränkung $h \upharpoonright S$ ist **injektiv**, d. h. für jedes $j \in [m]$ existiert höchstens ein $x \in S$ mit $h(x) = j$.

Beispiel: Auf $S_1 = \{\text{Februar}^{(1)}, \text{Maerz}^{(4)}, \text{Mai}^{(8)}, \text{Juni}^{(0)}, \text{Juli}^{(11)}, \text{Dezember}^{(12)}\}$ ist die Funktion h von oben injektiv, aber nicht auf $S_2 = \{\text{Januar}^{(0)}, \text{Mai}^{(8)}, \text{Juli}^{(11)}, \text{November}^{(8)}\}$.

Sprechweise: h **perfekt für** S $:\Leftrightarrow h \upharpoonright S$ injektiv.

Falls h perfekt für S :

Speichere x bzw. (x, r) in Tabellenplatz $T[h(x)]$, für $x \in S$.

lookup(x):

- Berechne $j = h(x)$.
(Dies sollte **effizient** möglich sein.)
- Prüfe, ob in $T[j]$ Schlüssel x steht.
Falls ja: Inhalt (x, r) von $T[j]$ ausgeben, sonst „ $x \notin S$ “.

Kollision: $x, y \in S$, $x \neq y$, $h(x) = h(y)$.

Leider sind Kollisionen „praktisch unvermeidlich“.

(Ausnahme: Speziell auf S abgestimmte Hashfunktion h , fortgeschrittene Konstruktionen.)

Fall: h „rein zufällig“, $|S| = n$, $S = \{x_1, \dots, x_n\}$. D. h.:

Uniformitätsannahme (UF*):

Werte $h(x_1), \dots, h(x_n)$ in $[m]^n$ sind „rein zufällig“:
Jeder der m^n Vektoren $(j_1, \dots, j_n) \in [m]^n$
kommt mit derselben Wahrscheinlichkeit $1/m^n$
als $(h(x_1), \dots, h(x_n))$ vor.

(UF*) tritt z. B. (näherungsweise) ein, wenn

- (i) U endlich ist und
- (ii) h das Universum in gleich große Teile zerlegt, d. h.*

$$|h^{-1}(\{j\})| = \lfloor |U|/m \rfloor \quad \text{oder} \quad = \lceil |U|/m \rceil, \quad \text{für } j = 0, \dots, m-1, \quad \text{und}$$

- (iii) $S \subseteq U$ eine rein zufällige Menge der Größe n ist. (**(!) Sehr idealisierend!**)

* $\lfloor y \rfloor$ (bzw. $\lceil y \rceil$): größte (kleinste) ganze Zahl \leq (bzw. \geq) y .

Mit (UF*):

$$\begin{aligned}\Pr(\text{keine Kollision}) &= \Pr(h \text{ ist injektiv auf } \{x_1, \dots, x_n\}) = \\ &= \frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdot \dots \cdot \frac{m-n+1}{m} = \dots\end{aligned}$$

(Für $i = 1, \dots, n$ ist die Wahrscheinlichkeit, dass $h(x_i)$ in $[m] - \{h(x_1), \dots, h(x_{i-1})\}$ liegt, genau $\frac{m-i+1}{m}$.)

$$\begin{aligned}\dots &= 1 \cdot \left(1 - \frac{1}{m}\right) \cdot \left(1 - \frac{2}{m}\right) \cdot \dots \cdot \left(1 - \frac{n-1}{m}\right) \\ &< e^0 \cdot e^{-\frac{1}{m}} \cdot e^{-\frac{2}{m}} \cdot \dots \cdot e^{-\frac{n-1}{m}} = e^{-\frac{n(n-1)}{2m}}.\end{aligned}$$

(denn für $x \neq 0$ ist $1 + x < e^x$), also ...

$$\Pr(h \text{ injektiv auf } S) < e^{-\frac{n(n-1)}{2m}}.$$

Wunsch: Linearer Platzverbrauch, also

$$\text{„Auslastungsfaktor“ } \alpha := \frac{n}{m}$$

nicht kleiner als eine Konstante α_0 , z. B. $\alpha \geq \frac{1}{2}$ stets. Dann:

$$\Pr(h \text{ injektiv auf } S) \leq e^{-\alpha_0(n-1)/2}.$$

Winzig für nicht ganz kleine n ! D. h.: Falls h nicht auf S abgestimmt ist, kommen Kollisionen praktisch immer vor.

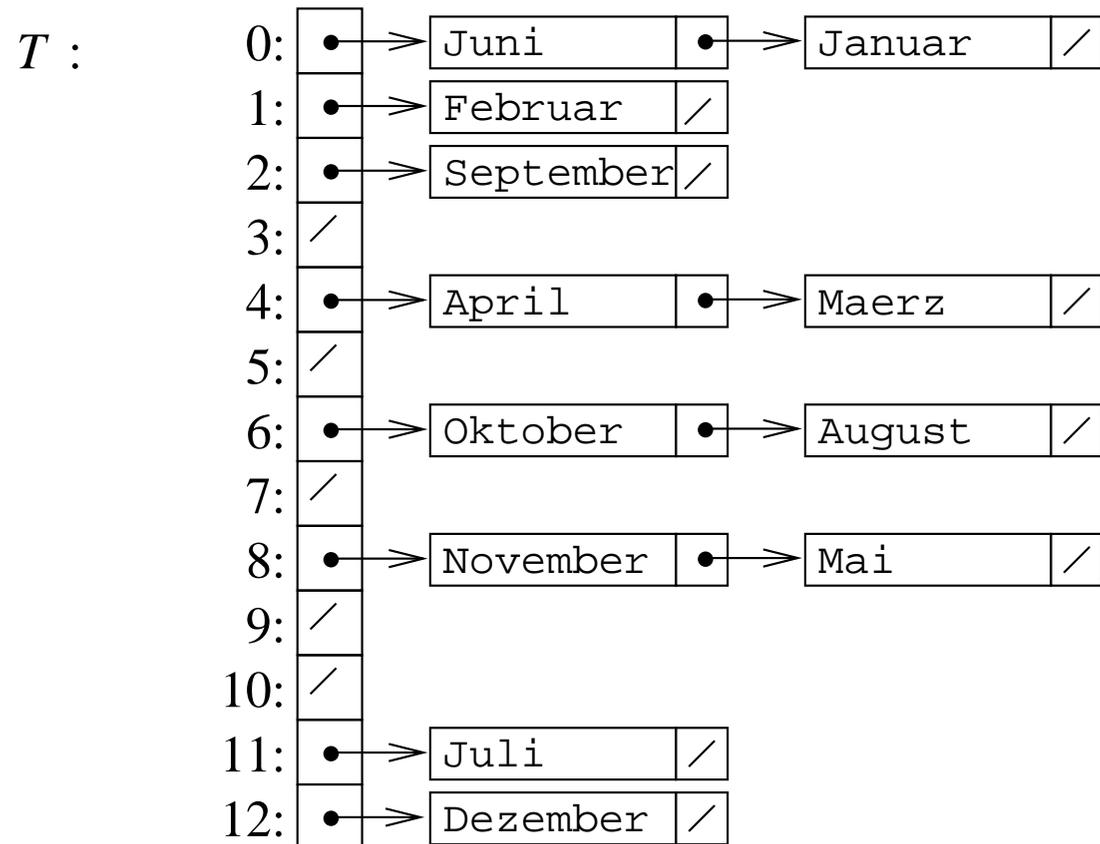
→ **Kollisionsbehandlung** ist notwendig.

Vorlesungsvideo:

Hashing mit verketteten Listen

5.2 Hashing „mit verketteten Listen“

Beispiel:



5.2 Hashing „mit verketteten Listen“

engl.: **chained hashing**

Auch: „**offenes Hashing**“, weil man Platz außerhalb von T benutzt.

In $T[0..m-1]$: **Zeiger** auf die Anfänge
von m **einfach verketteten linearen Listen** L_0, \dots, L_{m-1} .

(Oder auch: Zeiger auf „kleine“ Arrays/Vektoren mit variabler Länge.)

Liste L_j enthält die Einträge mit Schlüsseln x aus Behälter/Bucket

$$B_j = \{x \in S \mid h(x) = j\}.$$

Bemerkung: Hier, als Unterstruktur, sind lineare Listen perfekt geeignet!

Wir implementieren die Wörterbuchoperationen: **Algorithmen(skizzen)**.

empty(m): Erzeugt Array $T[0..m-1]$ mit m NULL-Zeigern/
NULL-Referenzen **und legt $h: U \rightarrow [m]$ fest.**

Kosten: $\Theta(m)$.

Beachte: Man wählt h , ohne S zu kennen!

```
lookup(x):  Berechne  $j = h(x)$ ;  
           suche Schlüssel  $x$  durch Durchlaufen der Liste  $L_j$  bei  $T[j]$ ;  
           falls Eintrag  $(x, r)$  gefunden: return  $r$ ;  
           // erfolgreiche Suche  
           sonst: return „nicht vorhanden“  
           // erfolglose Suche
```

Kosten/Zeit: $O(1 + \text{Zahl der Schlüsselvergleiche } „x = y?“)$

... falls die Auswertung von $h(x)$
und ein Schlüsselvergleich Zeit $O(1)$ kostet.

Erfolglose Suche: Exakt $|B_j|$ Schlüsselvergleiche.

Erfolgreiche Suche: Höchstens $|B_j|$ Schlüsselvergleiche.

insert(x, r): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;
falls gefunden: ersetze Daten bei x durch r ;
// **erfolgreiche Suche**
sonst: füge (x, r) in Liste $T[j]$ ein.
// **erfolglose Suche, Normalfall!**

delete(x): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;
falls gefunden: lösche Eintrag $(x, f(x))$ aus L_j ;
// **erfolgreiche Suche, Normalfall!**
sonst: tue nichts. // **erfolglose Suche**

Kosten jeweils: wie bei *lookup*.

Ziel: Analysiere **erwartete Kosten** für *lookup*, *insert*, *delete*.

Hierfür genügt: Analysiere **erwartete Anzahl der Schlüsselvergleiche**.

Vorausgesetzt: Irgendeine Quelle für „**Zufall**“. (Dazu später mehr.)

Pr(A): Wahrscheinlichkeit für **Ereignis** A .

E(X): Erwartungswert der **Zufallsvariablen** X .

Abgeschwächte „Uniformitätsannahme (UF₂)“:

$$\text{Für je 2 Schlüssel } x \neq z \text{ in } U \text{ gilt: } \Pr(h(x) = h(z)) \leq \frac{1}{m}.$$

Später: Wann ist diese Annahme erfüllt?

Wenn die Hashwerte rein zufällig sind, gilt: $\Pr(h(x) = h(z)) = \frac{1}{m}$.

Szenario: Menge $S = \{x_1, \dots, x_n\}$ von n Schlüsseln in Struktur gespeichert.
Nun wird nach $y \in U$ gesucht.

$\alpha = \frac{n}{m}$: der (aktuelle) **Auslastungsfaktor**.

Erfolglose Suche: $y \notin S$.

Definiere **Zufallsgrößen/Zufallsvariable**, für alle $x, z \in U$:

$$Y_{x,z} := \begin{cases} 1, & \text{falls } h(x) = h(z), \\ 0, & \text{falls } h(x) \neq h(z). \end{cases}$$

Nach (UF₂): $\mathbf{E}(Y_{x,z}) = \mathbf{Pr}(h(x) = h(z)) \leq \frac{1}{m}$.

Anzahl der Schlüsselvergleiche/Listenlänge =

$$|B_{h(y)}| = |\{x \in S \mid h(x) = h(y)\}| = \sum_{x \in S} Y_{x,y}.$$

Eine Zufallsvariable!

Mit „Linearität des Erwartungswertes“:

$$\mathbf{E}(|B_{h(y)}|) = \sum_{x \in S} \mathbf{E}(Y_{x,y}) \stackrel{(\text{UF}_2)}{\leq} \sum_{x \in S} \frac{1}{m} = \frac{n}{m} = \alpha.$$

Satz 5.2.3

Bei Hashing mit verketteten Listen gilt unter der Annahme (UF₂):

Die **erwartete Anzahl** von Schlüsselvergleichen bei **erfolgloser** Suche ist höchstens α , bei **erfolgreicher** Suche höchstens $1 + (n - 1)/m < 1 + \alpha$.

(Analog; Summand $\mathbf{E}(Y_{y,y}) = 1$.)

Die **erwartete Zeit** für eine erfolglose/erfolgreiche Suche ist $O(1 + \alpha)$.

Fazit: Wenn $\alpha \leq \alpha_1$, für α_1 konstant (z.B. $\alpha_1 = 2$), dann ist die **erwartete Zeit** für eine Suche $O(1)$.

Verdoppelungsstrategie

... hilft, die Bedingung $\alpha \leq \alpha_1$ zu erzwingen, auch wenn anfangs die Schlüsselanzahl unbekannt ist.

(Analog zu Verdoppelungsstrategie bei Stacks, Kap. 2)

empty(m): Anfangs-Tabellengröße m oder

empty(): Anfangs-Tabellengröße m_0 (fester Wert)

Variable *load* enthält aktuelle Anzahl von Schlüsseln.

empty(m): $\text{load} \leftarrow 0,$

insert(x, r) mit x neu: $\text{load} \leftarrow \text{load} + 1,$

delete(x) mit x vorhanden: $\text{load} \leftarrow \text{load} - 1.$

Verdoppelungsstrategie (Fortsetzung)

Wenn nach $insert(x, r)$ (mit x neu) der Wert $load$ die Grenze $\alpha_1 m$ überschreitet:

- Erzeuge **neue Tabelle** T_{neu} der Größe $2m$.
- Erzeuge **neue Hashfunktion** $h_{neu}: U \rightarrow [2m]$.
- Übertrage Einträge aus den alten Listen in neue Listen zu Tabelle T_{neu} , gemäß neuer Hashfunktion. Dabei Einfügen **am Listenanfang ohne Suchen**.
(Wissen: Alle Schlüssel sind verschieden!)
Kosten: $O(m)$ im schlechtesten Fall.
- Benenne T_{neu} in T und h_{neu} in h um.

Satz 5.2.4

Sei α_1 fest. Wenn die verwendeten Hashfunktionen (UF_2) erfüllen, dann gilt für das skizzierte Verfahren für Hashing mit Listen mit Verdoppelungsstrategie:

- (a) Die erwartete Zeit für erfolglose bzw. erfolgreiche Suche ist $O(1 + \alpha_1)$.
- (b) Die Gesamtkosten für sämtliche Verdopplungen sind durch $O(\tilde{n})$ (schlechtester Fall) beschränkt; dabei ist \tilde{n} die maximale Anzahl der jemals gleichzeitig in der Datenstruktur vorhandenen Schlüssel (maximaler Wert in load).
- (c) Der Gesamtplatzbedarf ist $O(\tilde{n})$.

(Zu (b): **Analyse** der Verdoppelungsstrategie wie bei Stacks.)

Vorlesungsvideo:

Hashfunktionen

5.3 Hashfunktionen

5.3.1 Einfache Hashfunktionen

Kriterien für Qualität:

- 1) **Zeit für Auswertung** von $h(x)$ (*Schnell!*)
- 2) **Wahrscheinlichkeit für Kollisionen**
(unter „realistischen Annahmen“ über Schlüsselmenge S).
(*Klein*, am besten $\leq c/m$ für c konstant.)

Für 1) entscheidend: **Schlüsselformat**.

1. Fall: Schlüssel sind natürliche Zahlen: $U \subseteq \mathbb{N}$.

Divisionsrestmethode:

$$h(x) = x \bmod m$$

Sehr einfach, recht effizient.

Nachteile:

- Unflexibel: Nur eine feste Funktion für eine Tabellengröße.
- Bei strukturierten Schlüsselmengen (z. B. aus Strings übersetzte Schlüssel) deutlich erhöhtes Kollisionsrisiko.

Für Produktionssysteme darf diese Methode nicht benutzt werden!

Multiplikationsmethode: $U \subseteq \mathbb{N}$. Idee: Für $0 < \vartheta < 1$ (idealisiert: ϑ **irrational**)

$h(x) := \lfloor ((\vartheta \cdot x) \bmod 1) \cdot m \rfloor$. (Dabei: $(\vartheta \cdot x) \bmod 1 = \vartheta \cdot x - \lfloor \vartheta \cdot x \rfloor \in [0, 1)$.)

(Gebrochener Anteil von ϑx wird auf Wertebereich $[0, m)$ „aufgeblasen“, dann auf ganze Zahl abgerundet. Für $\theta = \frac{1}{2}(\sqrt{5} - 1) \approx 1,618$ weiß man: $x, x + 1, \dots, x + n - 1$ werden gut verteilt.)

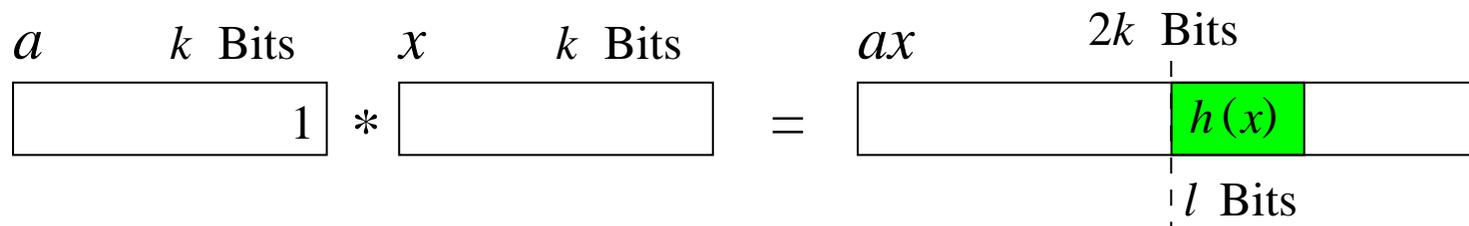
Diskrete Multiplikationsmethode: $U = [2^k]$ und $m = 2^l$.

$$h_a(x) = (ax \bmod 2^k) \operatorname{div} 2^{k-l}.$$

mit $a \in [2^k]$ ungerade und z. B. $a/2^k \approx \frac{1}{2}(\sqrt{5} - 1) = 0,618\dots$

Oder: $a \in [2^k]$ ungerade, **zufällig**.

Sehr effiziente Auswertung ohne Division:



(Wenn $k =$ Wortbreite w : Multiplikation + Shift.)

2. Fall: Schlüssel sind Strings/Wörter: $U \subseteq \text{Seq}(\Sigma)$

Σ : „**Alphabet**“ – endliche nichtleere Menge.

Seq(Σ): Menge aller endlichen Folgen von Elementen von Σ : „Wörter über Σ “.

Beispiel: $\Sigma = \text{ISO-8859-1-Alphabet} \cong \{0, 1\}^8 \cong \{0, 1, \dots, 255\}$.

(Siehe https://de.wikipedia.org/wiki/ISO_8859-1#ISO/IEC_8859-1.)

Schlüssel: $x = (c_1, \dots, c_r) = c_1 \dots c_r$ mit $c_1, \dots, c_r \in \Sigma$.

Möglich, aber nicht zu empfehlen:

Transformation von Schlüssel x in Zahl $\hat{x} = n(x)$ als neuen Schlüssel, dann: $x \mapsto h(\hat{x})$ für Hashfunktion h auf Zahlen.

- Kollisionen $n(x) = n(y)$ nie auflösbar!
- Selbst wenn n injektiv: Ineffiziente Auswertung von $h(\hat{x})$.

Besser: An Schlüsselformat $x = (c_1, \dots, c_r)$ angepasste Hashfunktionen.

Lineare Funktionen über Körper $\mathbb{Z}_p = [p] = \{0, \dots, p-1\}$:

$\Sigma \subseteq [p]$ für p Primzahl, $m = p$. (Beispiel: ISO-8859-1 $\hat{=}$ [256] \subseteq [p] = [1009].)

Wähle Koeffizienten $a_1, \dots, a_r \in [p]$.

$$h(c_1 \cdots c_r) = \left(\sum_{i=1}^r a_i \cdot c_i \right) \bmod p.$$

Vorteile:

- Rechnung nur mit Zahlen $< p^2 = m^2 \dots$ (wenn man nach jeder Multiplikation/Addition Rest modulo p bildet);
- sehr effizient;
- theoretisch nachweisbares gutes Verhalten, wenn a_1, \dots, a_r aus $[p]$ **zufällig** gewählt werden.

5.3.2 Anspruchsvollere Hashfunktionen

Qualitätskriterien:

- 1) Zeit für **Auswertung** von $h(x)$ (*klein!*)
- 2) geringe **Wahrscheinlichkeit** für **Kollisionen**
ohne Annahmen über die Schlüsselmenge S (ähnlich der zufälligen Situation!)
- 3) Effizientes **Generieren** einer neuen Hashfunktion
(benötigt z. B. bei der Verdoppelungsstrategie)
- 4) **Platzbedarf** für die Speicherung von h (Programmtext, Parameter)
(*gering* gegenüber $n!$)

Universelles Hashing (Idee): Bei der Festlegung der Hashfunktion h wird (kontrolliert) **Zufall** eingesetzt („Randomisierter Algorithmus“).

(Bei Programmierung: „Pseudozufallszahlengenerator“, in jeder Prog.-Sprache bereitgestellt.)

Analyse beruht auf dadurch **vom Algorithmus erzwungenen** Zufallseigenschaften von h , nicht auf Annahmen über S .

Beispiel: Alphabet $\Sigma = [s]$, $U \subseteq [s]^r$, und $m = p \geq s$ eine **Primzahl**.

[Konkret: $U = (\text{ISO 8859-1})^{20} = \{0, \dots, 255\}^{20}$, $s = 256$, $m = p = 1009$.]

a_1, \dots, a_r seien (uniform) **zufällig gewählt** aus $[p]$. (In $\text{empty}(p)$.)

Definiere

$$h(\underbrace{c_1 \cdots c_r}_x) = (a_1 \cdot c_1 + \cdots + a_r \cdot c_r) \bmod p.$$

Proposition 5.3.1

In der beschriebenen Situation gilt für $x, z \in U$ mit $x \neq z$:

$$\Pr(h(x) = h(z)) = \frac{1}{p}.$$

(Zufall in der Wahrscheinlichkeit stammt aus dem Algorithmus.) **(UF₂) ist erfüllt!**

Beweis: Sei $x = (c_1, \dots, c_r)$, $z = (d_1, \dots, d_r)$.

Rechne im **Körper** $\mathbb{Z}_p = \mathbb{Z}/p\mathbb{Z}$ „=" $[p]$ (also modulo p).

Es gibt p^r viele Möglichkeiten für $a = (a_1, \dots, a_r)$.

Wie viele davon erfüllen

$$a_1c_1 + \dots + a_rc_r = a_1d_1 + \dots + a_rd_r, \quad \text{d. h. } a_1(c_1 - d_1) + \dots + a_r(c_r - d_r) = 0 ?$$

Beispielsweise sei $c_r - d_r \neq 0$.

Für jede Wahl von a_1, \dots, a_{r-1} gibt es genau ein passendes a_r , nämlich

$$a_r = -(a_1(c_1 - d_1) + \dots + a_{r-1}(c_{r-1} - d_{r-1})) \cdot (c_r - d_r)^{-1}.$$

(Wir rechnen modulo p , mit Inversen im Körper \mathbb{Z}_p .) Also ist die Wahrscheinlichkeit, einen passenden Vektor (a_1, \dots, a_r) zu wählen, genau $\frac{p^{r-1}}{p^r} = \frac{1}{p}$. □

Variante: $U \subseteq [p]$, p Primzahl.

Tabellengröße $m \leq p$ beliebig.

Wähle a aus $\{1, \dots, p-1\}$ (uniform) **zufällig**.

Definiere

$$h(x) = ((a \cdot x) \bmod p) \bmod m.$$

Proposition 5.3.2

In dieser Situation gilt für $x, z \in U$ mit $x \neq z$:

$$\Pr(h(x) = h(z)) \leq \frac{2}{m}.$$

(D. h. eine Abschwächung von (UF_2) ist erfüllt.)

Beweis (nicht prüfungsrelevant): Druckfolien.

Beweis (nicht prüfungsrelevant): Für $a \in \{1, \dots, p-1\}$ setze:

$$h_a(x) := (ax \bmod p) \bmod m, \text{ für } x \in U.$$

Seien $x, z \in U$ verschieden mit $h_a(x) = h_a(z)$. Setze $y := x - z$.

1. Fall: $ax \bmod p > az \bmod p$.

Dann ist $ax \bmod p - az \bmod p = ay \bmod p \in \{1, \dots, p-1\}$ durch m teilbar.

2. Fall: $ax \bmod p < az \bmod p$.

Dann ist $ax \bmod p - az \bmod p = ay \bmod p - p \in \{-1, \dots, -(p-1)\}$ durch m teilbar.

Also gilt: $ay \bmod p \in \{i \cdot m \mid 1 \leq i \leq \frac{p-1}{m}\} \cup \{p - i \cdot m \mid 1 \leq i \leq \frac{p-1}{m}\}$.

Weil $y \neq 0$, ist $ay \bmod p$ uniform zufällig in $\{1, \dots, p-1\}$ verteilt. – Also:

$$\begin{aligned} \Pr(h_a(x) = h_a(z)) &\leq \Pr(ay \bmod p \in \{i \cdot m \mid 1 \leq i \leq \frac{p-1}{m}\} \cup \{p - i \cdot m \mid 1 \leq i \leq \frac{p-1}{m}\}) \\ &\leq \frac{2(p-1)/m}{p-1} = \frac{2}{m}. \end{aligned}$$

□

Proposition 5.3.3

Wenn man beim **multiplikativen Hashing** (wobei $U = [2^k]$, $m = 2^l$), mit

$$h(x) = (\mathbf{a} \cdot x \bmod 2^k) \operatorname{div} 2^{k-l}$$

als **a** eine **zufällige ungerade** Zahl in $[2^k]$ wählt, dann gilt für beliebige $x, z \in U$, $x \neq z$:

$$\Pr(h(x) = h(z)) \leq \frac{2}{2^l} = \frac{2}{m}.$$

D. h.: Eine Abschwächung von (UF_2) ist erfüllt.

(*Beweis*: Originalliteratur [M. D., T.Hagerup, J.Katajainen, M.Penttonen, A Reliable Randomized Algorithm for the Closest-Pair Problem, J. Algorithms **25**(1): 19-51 (1997)].)

Tabellenhashing [\[Zobrist, 1970\]](#)

Universum: $U = \Sigma^r$, mit $\Sigma = \{0, \dots, s - 1\}$.

$m = 2^l$ mit $l \leq w$ (w : Wortlänge, z. B. $w = 32$).

Wertebereich: $[m]$, entspricht $\{0, 1\}^l$ (via Binärdarstellung!).

Repräsentation der Hashfunktion h :

Array $A[1..r, 0..s - 1]$ mit Einträgen aus $\{0, 1\}^w$.

$h(c_1 \cdots c_r) =$ die ersten l Bits von $A[1, c_1] \oplus \cdots \oplus A[r, c_r]$

Dabei: \oplus ist bitweises XOR, auf ganzes Wort angewendet.
(Maschinenoperation – effizient!)

Beispiel: $\Sigma = \{0, 1, \dots, 7\}$, $r = 6$ (1 Schlüssel = 6 Oktalziffern), $w = 4$, $l = 3$.

Array A:

	Buchstaben in Σ (Oktalziffern)							
	0	1	2	3	4	5	6	7
Position $i = 1$	0010	1001	0011	<u>1010</u>	0111	0000	1101	0111
2	0011	1000	1111	0100	1001	<u>1101</u>	0110	0110
3	1000	0110	0101	<u>1001</u>	1110	1111	0000	1110
4	1101	0100	1110	0111	1101	0100	1010	<u>0110</u>
5	1100	1001	1100	1011	<u>0101</u>	1011	1110	0000
6	0110	1101	1001	<u>1111</u>	1000	1110	0011	0111

$h(353743)$

= die ersten 3 Bits von $1010 \oplus 1101 \oplus 1001 \oplus 0110 \oplus 0101 \oplus 1111$

= die ersten 3 Bits von $0010 = 001$.

Nachteil: Array $A[1..r, 0..s-1]$ benötigt **Platz** $r \cdot s$ (Wörter).

Beispiel: $r = 20$ ISO-8859-1-Zeichen pro Wort, $\Sigma = \{0, 1\}^8$, $w = 32$.

$A[1..r, 0..s-1]$ benötigt

$rs(w/8) = 20 \cdot 256 \cdot 4 = 20480$ Bytes, d. h. 20 KB.

Sinnvoll nur für **große** Hashtabellen.

Platzsparend, etwas langsamer:

$\Sigma = \{0, 1\}^4 \hat{=} \{0, 1, \dots, 9, A, \dots, F\}$. („Tetraden“ bzw. Hexadezimalziffern.)

Platz im Beispiel: $rs(w/8) = 40 \cdot 16 \cdot 4 = 2560$ Bytes.

Vorteile von Tabellenhashing:

- **Extrem schnelle Auswertung** (\oplus ebenso schnell wie $+$).
- Dasselbe A kann für $m = 2^4, 2^5, 2^6, \dots, 2^w$ benutzt werden.
Ideal für die Kombination mit der Verdoppelungsstrategie: Verdopple
Tabellengröße $\hat{=}$ erhöhe l um 1; Einträge in A bleiben unverändert.
- Beweisbar **gutes Verhalten**, wenn Einträge von A zufällig.

Proposition 5.3.4

Wenn die Einträge in Array $A[1..r, 0..s-1]$ uniform **zufällig** gewählt werden, dann gilt für $x, y, z \in U$ mit x, y, z verschieden und $a, b, c \in \{0, 1\}^l$ beliebig:

$$\Pr(h(x) = a \wedge h(y) = b \wedge h(z) = c) = \frac{1}{m^3}.$$

Insbesondere gilt (UF_2) .

Beweis: Übung.

Leichte Modifikation des Beweises von Satz 5.2.3. liefert:

Proposition 5.3.5

Sei c konstant. Wenn $\Pr(h(x) = h(z)) \leq \frac{c}{m}$, für beliebige $x, z \in U$, $x \neq z$, und wenn wir **Hashing mit verketteten Listen** benutzen, dann ist die **erwartete** Zahl von Schlüsselvergleichen

- höchstens $c \cdot \alpha$ bei der **erfolglosen Suche** und
- höchstens $1 + c \cdot \alpha$ bei der **erfolgreichen Suche**.

Wenn $\alpha \leq \alpha_1$ sichergestellt ist, für α_1 konstant (Verdoppelungsstrategie): Erwartete Suchzeit ist **$O(1)$** .

Stichwort: „Universelles Hashing“: siehe Druckfolien.

Universelles Hashing (Formales Konzept)

SS 2021: Nicht prüfungsrelevant.

Eine Menge \mathcal{H} von Hashfunktionen, d.h. $\mathcal{H} \subseteq \{h \mid h: U \rightarrow [m]\}$, heißt **c-universelle** Klasse von Hashfunktionen, wenn für beliebige $x, z \in U$ mit $x \neq z$ gilt:

$$\frac{|\{h \in \mathcal{H} \mid h(x) = h(z)\}|}{|\mathcal{H}|} \leq \frac{c}{m}.$$

In anderen Worten:

... wenn für für beliebige $x, z \in U$ mit $x \neq z$ und für zufällig gewähltes h aus \mathcal{H} gilt:

$$\Pr(h(x) = h(z)) \leq \frac{c}{m}.$$

Beispiele:

- $U \subseteq [p]$, p Primzahl, Tabellengröße $m \leq p$ beliebig.

$$h_a(x) = ((a \cdot x) \bmod p) \bmod m, \text{ für } a \in \{1, \dots, p-1\}.$$

Dann ist $\mathcal{H}_{p,m} = \{h_a \mid 1 \leq a < p\}$ eine 2-universelle Klasse.

- $U = (\text{ISO 8859-1})^r \hat{=} \{0, \dots, 255\}^r$, $m = p > 256$ Primzahl.

$$h_{a_1, \dots, a_r}(c_1 \dots c_r) = \left(\sum_{i=1}^r a_i c_i \right) \bmod p, \text{ für } c_1 \dots c_r \in U.$$

Dann ist $\mathcal{H}_{p,r} = \{h_{a_1, \dots, a_r} \mid a_1, \dots, a_r \in [p]\}$ eine 1-universelle Klasse.

Beispiele:

- **Multiplikatives Hashing** mit zufälligem (ungeradem) Faktor a führt zu einer 2-universellen Klasse.
- **Tabellenhashing** mit zufälligen Tabelleneinträgen führt zu einer 1-universellen Klasse.

Die obige **Proposition 5.3.5** liest sich nun so:

Wenn \mathcal{H} c -universell ist, wenn man Hashing mit verketteten Listen benutzt und (in $empty(m)$) ein h aus \mathcal{H} **zufällig** wählt,

dann gelten die oberen Schranken $c \cdot \alpha$ bzw. $1 + c \cdot \alpha$ für die erwartete Anzahl der Schlüsselvergleiche bei **erfolgreicher** bzw. **erfolgreicher** Suche.

Vorlesungsvideo:

Geschlossenes Hashing mit Sondierungsstrategien

5.4 Geschlossenes Hashing

oder „**offene Adressierung**“: Kollisionsbehandlungs-Methode.

Idee: Für jeden Schlüssel $x \in U$ gibt es eine

Sondierungsfolge $h(x, 0), h(x, 1), h(x, 2), \dots$ in $[m]$.

Beim **Einfügen** von x und **Suchen** nach x werden die Zellen von T in dieser Reihenfolge untersucht, bis eine leere Zelle oder der Eintrag x gefunden wird. –

Günstig:

(*) $h(x, 0), h(x, 1), \dots, h(x, m - 1)$ erreicht jede Zelle, d. h.:
 $(h(x, 0), h(x, 1), h(x, 2), \dots, h(x, m - 1))$
ist eine **Permutation** von $[m]$.

Wenn (*) gilt und mindestens eine Zelle in T leer ist, dann endet jede erfolgreiche Suche in einer leeren Zelle.

A – Lineares Sondieren $h(x, k) = (h(x) + k) \bmod m, k = 0, 1, 2, \dots$

Offensichtlich: (*) erfüllt.

Beispiel:

Nächste Tabelle: „Wunschplätze“ für die Schlüssel.

Übernächste Tabelle: Sequenzielles Einfügen in der natürlichen Reihenfolge gemäß „Linearem Sondieren“.

$$h(c_1 \dots c_r) = (\text{num}(c_3) + 11) \bmod 13$$

j	x
0	September ⁽⁰⁾
1	
2	Maerz ⁽²⁾ April ⁽²⁾
3	
4	August ⁽⁴⁾ Oktober ⁽⁴⁾
5	
6	Mai ⁽⁶⁾ November ⁽⁶⁾
7	
8	
9	Juli ⁽⁹⁾
10	Dezember ⁽¹⁰⁾
11	Januar ⁽¹¹⁾ Juni ⁽¹¹⁾
12	Februar ⁽¹²⁾

Tabelle T[0..12]:

Platz	Eintrag
0	Juni ⁽¹¹⁾
1	Maerz ⁽¹⁾
2	April ⁽¹⁾
3	September ⁽⁰⁾
4	August ⁽⁴⁾
5	
6	Mai ⁽⁶⁾
7	
8	
9	Juli ⁽⁹⁾
10	
11	Januar ⁽¹¹⁾
12	Februar ⁽¹²⁾

Tabelle T[0..12]:

Platz	Eintrag
0	Juni ⁽¹¹⁾
1	Maerz ⁽¹⁾
2	April ⁽¹⁾
3	September ⁽⁰⁾
4	August ⁽⁴⁾
5	Oktober ⁽⁴⁾
6	Mai ⁽⁶⁾
7	November ⁽⁶⁾
8	
9	Juli ⁽⁹⁾
10	Dezember ⁽¹⁰⁾
11	Januar ⁽¹¹⁾
12	Februar ⁽¹²⁾

empty(m): Lege leeres Array $T[0..m-1]$ an.

// Arrayeintrag: (Schlüssel,Wert (in R)) oder *leer*.

Wähle Hashfunktion h .

$T[j] \leftarrow \textit{leer}$, für $j \in [m]$.

$\textit{load} \leftarrow 0$, initialisiere $\textit{maxload}$ mit Wert $< m$.

insert(x, r): $j \leftarrow h(x)$;

Finde erstes l in der Folge $j, (j+1) \bmod m, (j+2) \bmod m, \dots$

mit $T[l].\textit{key} = x$ **oder** $T[l] = \textit{leer}$.

Im Fall $T[l].\textit{key} = x$: // **erfolgreiche Suche**, Update-Situation

$T[l].\textit{data} \leftarrow r$;

Im Fall $T[l] = \textit{leer}$: // **erfolglose Suche**, neuer Eintrag

Falls $\textit{load} = \textit{maxload}$: „Überlaufbehandlung“

// z. B. Übertragen der Einträge von T in neue, doppelt so große Tabelle

// Neustart von **insert**(x, r)

Sonst: $T[l] \leftarrow (x, r)$; $\textit{load}++$.

lookup(x): $j \leftarrow h(x)$;

Finde erstes l in der Folge $h(x, j)$, $j = 0, 1, 2, \dots$, mit $T[l].\text{key} = x$ **oder** $T[l] = \text{leer}$.

Im Fall $T[l].\text{key} = x$: **return** $T[l].\text{data}$; // **erfolgreiche Suche**

Im Fall $T[l] = \text{leer}$: **return** „nicht vorhanden“. // **erfolglose Suche**

Um die **Korrektheit** (des Ein-/Ausgabeverhaltens) zu zeigen, betrachten wir eine beliebige Operationenfolge und zeigen die folgende Invariante, für jeden Schlüssel x :

Invariante: Solange x nicht eingefügt wurde, gibt es in der Tabelle keinen Eintrag mit Schlüssel x . Nachdem x eingefügt wurde, gilt: Es gibt genau einen solchen Eintrag, in einer Zelle $T[i]$. Diese enthält immer den Datensatz r von der *letzten* Operation **insert**(x, r). Wenn $i = h(x, l)$, enthalten alle Zellen $T[h(x, j)]$, $0 \leq j < l$, einen Schlüssel $\neq x$. (Beweis durch Induktion über die ausgeführten Einfügungen, auf den Druckfolien.)

Korrektheit der Suche: Aus der Invarianten folgt sofort durch Betrachten der Algorithmen, dass **lookup**(x) einen Eintrag mit Schlüssel x genau dann findet, wenn es einen gibt, und dass dieser immer den Datensatz r enthält, der von der neuesten Operation **insert**(x, r) herrührt. (Details: Druckfolien.)

Beweis der Invarianten: Dass es keinen Eintrag mit Schlüssel x gibt, wenn x noch nicht eingefügt wurde, ist klar. Da Schlüssel nie gelöscht werden, sondern nur Datensätze überschrieben werden, bleibt ein einmal generierter Eintrag mit Schlüssel x immer erhalten.

Für den zweiten Teil benutzen wir Induktion. Betrachte die erste Einfügeoperation **insert**(x, r) für x . Die Suche nach x endet in einer leeren Zelle $T[i]$, mit $i = h(x, l)$, wobei alle Zellen $T[h(x, j)]$, $0 \leq j < l$, einen Schlüssel $\neq x$ enthalten. Hier wird (x, r) gespeichert. Wenn später in **insert**(x, r') erneut nach x gesucht wird, dann folgt aus der Invarianten und dem Suchverfahren, dass die eindeutig bestimmte Zelle $T[i]$ mit Schlüssel x gefunden wird. Also aktualisiert **insert**(x, r') den richtigen Datensatz (und erzeugt kein Duplikat).

Zur **Korrektheit der Suche**: Aus der Invarianten folgt, dass **lookup**(x) nur dann einen Datensatz zurückgibt, wenn x vorher eingefügt wurde. Wenn dies der Fall ist, stellt die Invariante sicher, dass **lookup**(x) die eindeutig bestimmte Zelle $T[i]$ mit Schlüssel x findet, und dass diese den aktuellen Datensatz für diesen Schlüssel enthält.

Satz 5.4.1 (1962 bewiesen von Donald E. Knuth (*1938, TAOCP, T_EX))

Bei linearem Sondieren in einer Tabelle der Größe m mit Schlüsselmenge $S = \{x_1, \dots, x_n\}$ gilt, mit festem $\alpha = \frac{n}{m}$ (und $n, m \rightarrow \infty$):

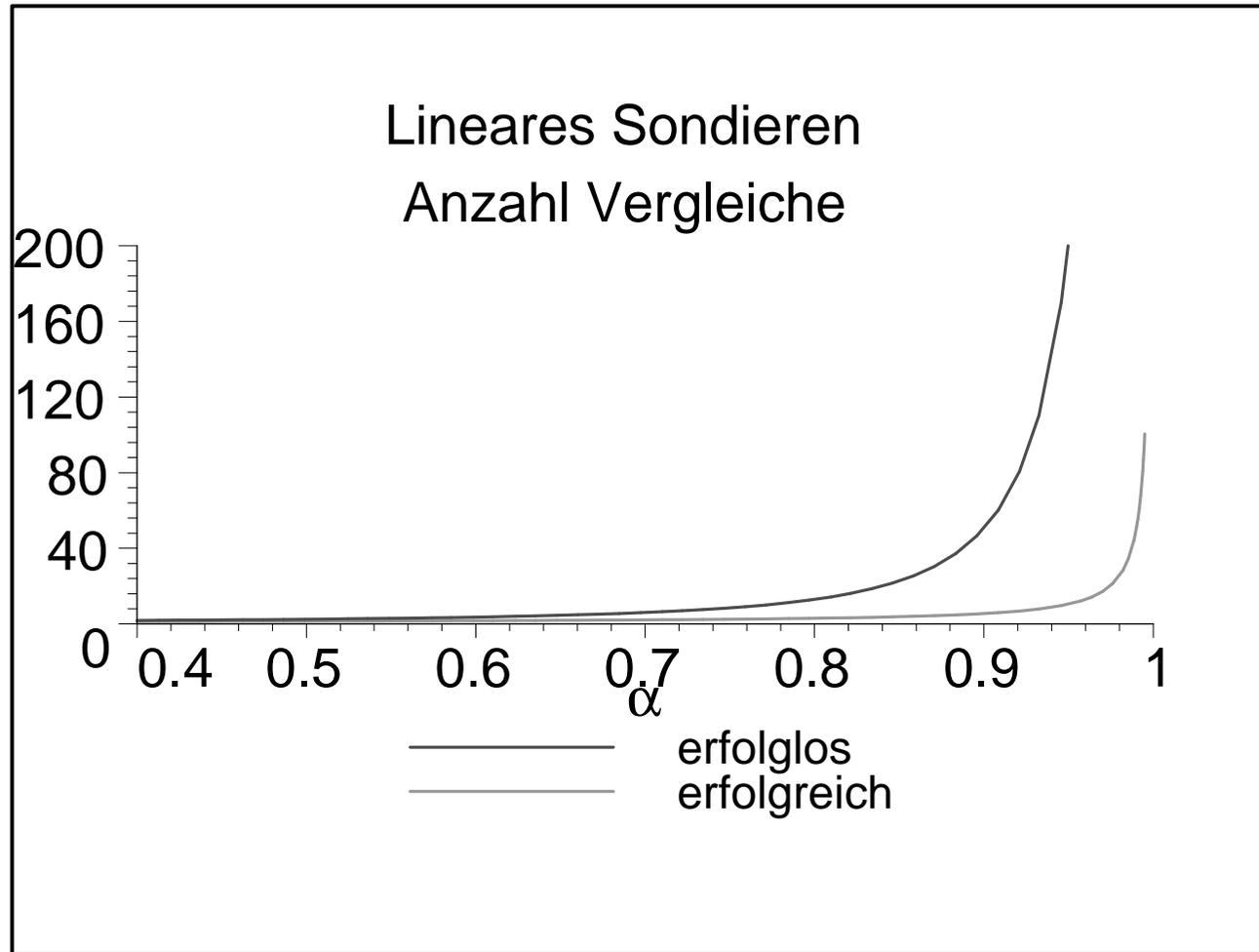
Die **erwartete** (nach (UF*): $h(x_1), \dots, h(x_n)$ **rein zufällig**) Anzahl von Schlüsselvergleichen (besuchte Arrayzellen) bei **erfolgloser Suche** nach $y \notin S$ ist

$$\approx \frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2} \right).$$

Die **erwartete** (nach (UF*)) **mittlere** (über x_1, \dots, x_n) Anzahl von Schlüsselvergleichen bei **erfolgreicher Suche** nach $y \in S$ ist

$$\approx \frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right).$$

Der erwartete Aufwand ist also **$O(1)$** , falls $\alpha \leq \alpha_0$ für eine Konstante $\alpha_0 < 1$.



Erwartete Anzahl von Schlüsselvergleichen bei linearem Sondieren:

	erfolglose Suche	erfolgreiche Suche (Mittel über x_1, \dots, x_n)
α	$\frac{1}{2} \cdot \left(1 + \frac{1}{(1-\alpha)^2}\right)$	$\frac{1}{2} \cdot \left(1 + \frac{1}{1-\alpha}\right)$
0,5	2,5	1,5
0,6	3,625	1,75
0,7	6,06	2,16
0,75	8,5	2,5
0,8	13	3
0,9	50,5	5,5
0,95	200,5	10,5
0,98	1250,5	25,5

Ergebnisse für lineares Sondieren:

Eine einzelne erfolglose Suche wird z. B. bei zu 90% gefüllter Tabelle durchaus teuer.

Bei vielen erfolglosen Suchen in einer nicht veränderlichen Tabelle (anwendungsabhängig!) sind diese Kosten nicht tolerierbar.

$\alpha \leq 0,6$ oder $\alpha \leq 0,7$ liefert jedoch akzeptable Suchkosten.

Die (über $x_i \in S$) **gemittelte** Einfüge-/Suchzeit ist dagegen sogar bei 90% gefüllter Tabelle erträglich.

!!!

Lineares Sondieren passt perfekt zu **Cache-Architekturen**

und ist deswegen auch in der Praxis sehr schnell.

(Bei einem Cachefehler wird eine ganze „Zeile“ in den Cache geholt, also mehrere aufeinanderfolgende Wörter aus T.)

Verdoppelungsstrategie bei linearem Sondieren:

Verdopple die Tabelle, sobald der Auslastungsfaktor $\alpha = n/m$ einen Wert von α_0 übersteigt – z.B. $\alpha_0 = 0,75$ oder $\alpha_0 = 0,8$.

Empfehlung: Kombiniere lineares Sondieren mit einer guten universellen Hashklasse, z. B. Tabellenhashing.

Nicht auf die Zufälligkeit von S vertrauen!

Alternative: **MurmurHash3** (32-Bit- oder 128-Bit-Werte).

<https://en.wikipedia.org/wiki/MurmurHash>

Lässt „seed“ zu, d. h. unterschiedliche Funktionen.

Bisherige experimentelle Untersuchungen: schnelle Auswertung, gute Verteilung.

Jedoch: Keine mathematische Aussage bekannt.

Achtung! Kryptographische Hashfunktionen (z.B. MD4, MD5, SHA-1, SHA-3, . . .) haben andere Anforderungen und **viel höhere Auswertezeiten!**

Sie sind für Anwendungen mit Hashtabellen nicht geeignet.

Andere Sondierungsfolgen sind möglich, zum Beispiel . . .

B – Quadratisches Sondieren

$h: U \rightarrow [m]$ sei beliebige Hashfunktion.

$$\begin{aligned}h(x, 0) &= h(x) \\h(x, 1) &= (h(x) + \mathbf{1}) \bmod m \\h(x, 2) &= (h(x) - \mathbf{1}) \bmod m \\h(x, 3) &= (h(x) + \mathbf{4}) \bmod m \\h(x, 4) &= (h(x) - \mathbf{4}) \bmod m \\h(x, 5) &= (h(x) + \mathbf{9}) \bmod m \\h(x, 6) &= (h(x) - \mathbf{9}) \bmod m \\&\vdots\end{aligned}$$

Allgemein:
$$h(x, k) = \left(h(x) + \lceil k/2 \rceil^2 \cdot (-1)^{k+1} \right) \bmod m, \text{ für } k = 0, 1, 2, \dots$$

(Man kann die Folge auch inkrementell nur mit Additionen/Subtraktionen modulo m berechnen.)

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

8, 9, 7, 12, 4, 17, 18, 5, 11, 14, 2, 6, 10, 0, 16, 15, 1, 13, 3.

Fakt 5.4.2

Ist m Primzahl, so dass $m + 1$ durch 4 teilbar ist, dann gilt bei der Verwendung von quadratischem Sondieren:

$$(*) \quad \{h(x, k) \mid 0 \leq k < m\} = [m].$$

(Beweis: Elementare Zahlentheorie.)

Beobachtung (empirisch):

Quadratisches Sondieren verhält sich sehr gut.

(Vor.: h gut verteilend, Auslastungsfaktor $\alpha \leq 0,9$.)

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1: U \rightarrow [m]$$

$$h_2: U \rightarrow [m - 1]$$

und setze, für $k = 0, 1, 2, \dots$:

$$h(x, k) := (h_1(x) + k \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \dots$$

Beispiel: $m = 13$, $h_1(x) = 3$, $1 + h_2(x) = 7$ liefert Sondierungsfolge

3, 10, 4, 11, 5, 12, 6, 0, 7, 1, 8, 2, 9.

Übung: Berechne die Folgenglieder **inkrementell** (Runden $k = 0, 1, 2, \dots$) ohne Multiplikationen und Divisionen („ $\dots \bmod m$ “). *Beispiel:* **3**, $3 + 7 = 10$, $(10 + 7) \bmod 13 = 4, \dots$

$h(x, k) := (h_1(x) + k \cdot (1 + h_2(x))) \bmod m$, für $k = 0, 1, 2, \dots$

Proposition 5.4.3

Wenn m Primzahl ist, dann gilt für **Doppel-Hashing**:

$$(*) \quad \{h(x, k) \mid 0 \leq k < m\} = [m].$$

Beweis: Sei $z = 1 + h_2(x)$. Dann: $1 \leq z \leq m - 1$. Wir müssen nur zeigen, dass die Abbildung $k \mapsto (h_1(x) + k \cdot z) \bmod m$ auf $\{0, 1, \dots, m - 1\}$ injektiv ist.

Dazu: Sei $h(x, k) = h(x, l)$ mit $0 \leq k, l < m$. Dann ist

$$(h_1(x) + k \cdot z) \bmod m = (h_1(x) + l \cdot z) \bmod m,$$

also $(l - k) \cdot z \equiv 0 \pmod{m}$, d.h. m teilt $(l - k) \cdot z$.

Da m Primzahl ist und $0 < z < m$ gilt, teilt m die Zahl $l - k$.

Weil $0 \leq |l - k| < m$, folgt $l - k = 0$, also $l = k$. □

Man kann beweisen: Wenn $h_1(x), h_2(x)$, $x \in S$, rein zufällig sind, verhält sich **Doppel-Hashing** sehr gut, nämlich im Wesentlichen wie „Uniformes Sondieren“.

Vorlesungsvideo:

Uniformes Sondieren

D – Uniformes Sondieren/Ideales Hashing

Keine Methode, sondern eine **Wahrscheinlichkeitsannahme**
für Verfahren mit offener Adressierung:

(IH):

$(h(x, 0), h(x, 1), \dots, h(x, m - 1))$ ist
eine **rein zufällige** Permutation von $[m]$,
unabhängig für jedes $x \in U$.

(Erinnerung: Es gibt $m!$ verschiedene Permutationen von $[m]$.)

Effekt von (IH): x_1, \dots, x_n seien gespeichert; y kommt neu hinzu.

Ganz egal was die Hashwerte von x_1, \dots, x_n und was $h(y, 0), \dots, h(y, k - 1)$ sind:
 $h(y, k)$ nimmt jeden Wert $j \in [m] - \{h(y, 0), \dots, h(y, k - 1)\}$
mit derselben Wahrscheinlichkeit $1/(m - k)$ an.

n : Anzahl der Schlüssel in $T[0..m-1]$; $\alpha = \frac{n}{m}$.

$T'_{m,n}$:= #(untersuchte Fächer bei **erfolgloser Suche**) (eine Zufallsvariable)

$C'_{m,n}$:= $\mathbf{E}(T'_{m,n})$.

$T_{m,n}$:= #(untersuchte Fächer bei **erfolgreicher Suche**), gemittelt über x_1, \dots, x_n
(eine Zufallsvariable).

$C_{m,n}$:= $\mathbf{E}(T_{m,n})$.

Satz 5.4.5

Unter der Annahme (IH) gilt, für $\alpha = n/m$:

(i) $C'_{m,n} = \frac{m+1}{m+1-n} = \frac{1}{1-\frac{n}{m+1}}$ ($< \frac{1}{1-n/m} = \frac{1}{1-\alpha}$).

Wenn α fest und $n, m \rightarrow \infty$, dann gilt $C'_{m,n} \rightarrow \frac{1}{1-\alpha}$.

(Immer wieder **rein zufällige** Positionen zu testen liefert Erfolgswahrscheinlichkeit $\frac{m-n}{m}$ in einem Versuch, also erwartete Versuchsanzahl $\frac{m}{m-n} = \frac{1}{1-\alpha}$.)

Beweis von (i): x_1, \dots, x_n sind gespeichert, $y \notin \{x_1, \dots, x_n\}$ wird gesucht.

In jedem Fall wird $T[h(y, 0)]$ untersucht, Kosten 1.

1. *Fall*: $T[h(y, 0)]$ ist leer.

Wahrscheinlichkeit: $\frac{m-n}{m}$. Weitere Kosten: 0.

2. *Fall*: $T[h(y, 0)]$ ist **nicht leer**.

Wahrscheinlichkeit: $\frac{n}{m}$. Weitere Kosten: $C'_{m-1, n-1}$.

(Suche in $m - 1$ Plätzen, von denen $n - 1$ besetzt sind.)

Rekursionsformel, für $0 \leq n < m$:

$$C'_{m,n} = \begin{cases} 1 & , \text{ falls } n = 0 \\ 1 + \frac{n}{m} \cdot C'_{m-1, n-1} & , \text{ falls } n \geq 1. \end{cases}$$

Wir haben: $C'_{m,n} = \begin{cases} 1 & , \text{ falls } n = 0 \\ 1 + \frac{n}{m} \cdot C'_{m-1,n-1} & , \text{ falls } n \geq 1. \end{cases}$ Also:

$$C'_{m,0} = 1;$$

$$C'_{m,1} = 1 + \frac{1}{m} \cdot 1 = \frac{m+1}{m};$$

$$C'_{m,2} = 1 + \frac{2}{m} \cdot \frac{m}{m-1} = \frac{m+1}{m-1};$$

$$C'_{m,3} = 1 + \frac{3}{m} \cdot \frac{m}{m-2} = \frac{m+1}{m-2};$$

und so weiter. Dies liefert:

$$C'_{m,n} = \frac{m+1}{m+1-n}.$$

(Formaler Beweis durch vollständige Induktion.)

□

Satz 5.4.5

Unter der Annahme (IH) gilt bei $\alpha = n/m$:

$$(i) \quad C'_{m,n} = \frac{m+1}{m+1-n} = \frac{1}{1-\frac{n}{m+1}}.$$

$$\text{Dies ist } < \frac{1}{1-n/m} = \frac{1}{1-\alpha}.$$

Wenn α fest und $n, m \rightarrow \infty$, dann gilt $C'_{m,n} \rightarrow \frac{1}{1-\alpha}$.

(ii) Für $T_{m,n} = \#(\text{untersuchte Fächer bei } \textbf{erfolgreicher Suche})$, gemittelt über x_1, \dots, x_n , gilt:

$$C_{m,n} = \mathbf{E}(T_{m,n}) < \frac{1}{\alpha} \cdot \ln\left(\frac{1}{1-\alpha}\right).$$

Beweis von (ii): Beachte: Suchaufwand für $x_i \in S$ = Einfügeaufwand für x_i .

Also Mittelung über $C'_{m,0}, C'_{m,1}, \dots, C'_{m,n-1}$:

$$\begin{aligned} C_{m,n} &= \frac{1}{n} \cdot \sum_{1 \leq i \leq n} \frac{m+1}{m+1-(i-1)} \\ &= \frac{m+1}{n} \cdot \left(\frac{1}{m+1} + \frac{1}{m} + \dots + \frac{1}{m+1-(n-1)} \right). \end{aligned}$$

Leicht zu sehen (Untersumme!):

$$\begin{aligned} \frac{1}{m+1} + \frac{1}{m} + \dots + \frac{1}{m+1-(n-1)} &\leq \int_{m+1-n}^{m+1} \frac{dt}{t} = \\ &= \ln(m+1) - \ln(m+1-n) = \ln\left(\frac{m+1}{m+1-n}\right). \end{aligned}$$

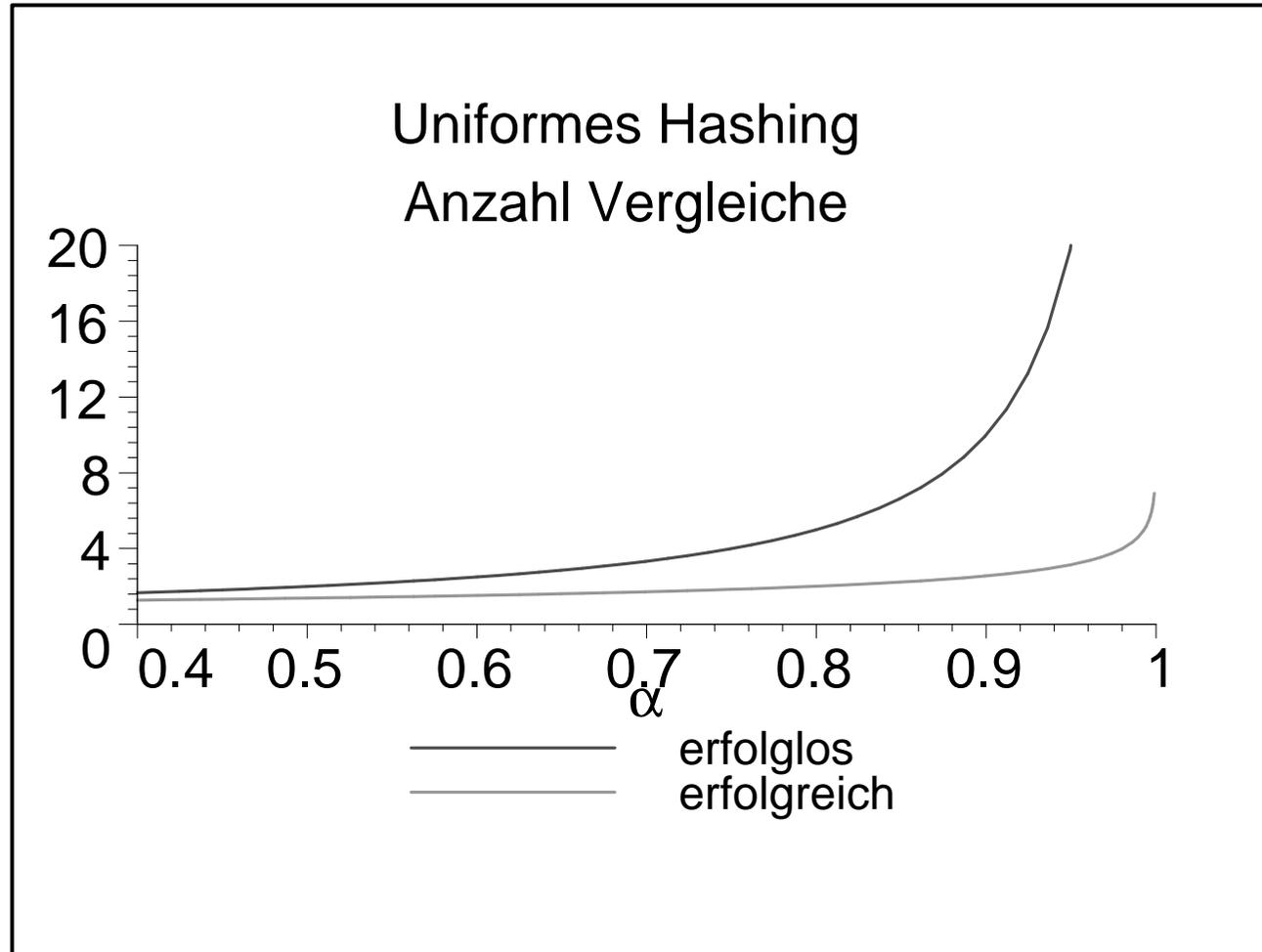
Also:

$$C_{m,n} \leq \frac{m+1}{n} \cdot \ln \left(\frac{1}{1 - \frac{n}{m+1}} \right) \stackrel{(!)}{<} \frac{m}{n} \cdot \ln \left(\frac{1}{1 - \frac{n}{m}} \right).$$

Für $\alpha = n/m$, α fest, $n, m \rightarrow \infty$, ist $C_{m,n} \approx \frac{1}{\alpha} \ln \left(\frac{1}{1-\alpha} \right)$. □

Für „ $\stackrel{(!)}{<}$ “ beweist man durch Kurvendiskussion: $t \mapsto \frac{1}{t} \cdot \ln \left(\frac{1}{1-t} \right)$ ist für $0 < t < 1$ monoton wachsend.

Bemerkung: Für $\alpha \nearrow 1$ geht $\frac{1}{\alpha} \cdot \ln \left(\frac{1}{1-\alpha} \right)$ **extrem langsam** gegen ∞ .



Erwartete Anzahl von Schlüsselvergleichen bei uniformem Hashing:

	erfolglose Suche	erfolgreiche Suche (Mittel über x_1, \dots, x_n)
α	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$
0,5	2	1,39
0,6	2,5	1,53
0,7	3,33	1,72
0,75	4	1,96
0,8	5	2,01
0,9	10	2,56
0,95	20	3,15
0,98	50	3,99
0,99	100	4,65

Vorlesungsvideo:

Löschen bei geschlossenem Hashing

Löschungen? Problem z. B. bei Quadratischem Sondieren.

T:

0:	April	(12)
1:	August	(1)
2:	Oktober	(1)
3:	Mai	(3)
4:	November	(3)
5:		
6:	Juli	(6)
7:	Juni	(8)
8:	Januar	(8)
9:	Februar	(9)
10:	September	(10)
11:	Dezember	(7)
12:	Maerz	(12)

Suchfolge „Dezember“ mit
 $h(\text{Dezember}) = 7: 7, 8, 6, 11.$

Lösche „Juli“!

Löschungen? Problem z. B. bei Quadratischem Sondieren.

T:

0:	April	(12)
1:	August	(1)
2:	Oktober	(1)
3:	Mai	(3)
4:	November	(3)
5:		
6:	!!!	
7:	Juni	(8)
8:	Januar	(8)
9:	Februar	(9)
10:	September	(10)
11:	Dezember	(7)
12:	Maerz	(12)

Suchfolge „Dezember“ mit
 $h(\text{Dezember}) = 7: 7, 8, 6, 11.$

Lösche „Juli“!

„Dezember“ wird nicht mehr gefunden.

Ausweg: Jede Zelle hat Statusvariable status (*voll*, *leer*, *gelöscht*).

Löschen: Zelle nicht auf *leer*, sondern auf *gelöscht* setzen (engl.: „*tombstone*“).

Suchen: Immer bis zur ersten *leer*en Zelle gehen.

Zellen, die *gelöscht* sind, dürfen mit einem neuen Schlüssel überschrieben werden.

Überlauf tritt **spätestens** ein, wenn die letzte *leer*e Zelle beschrieben werden würde.

Sinnvoller: Überlauf tritt ein, wenn die Anzahl der *voll*en und *gelöscht*en Zellen zusammen eine feste Schranke *bound* überschreitet.

Z.B. $m - 1$ oder $\alpha_1 m$, α_1 passend zum Sondierungsverfahren.

Alle Operationen bei offener Adressierung

empty(m):

Lege leeres Array $T[0..m-1]$ an.

Initialisiere alle Zellen $T[j]$ mit $(-, -, \text{leer})$.

Wähle Hashfunktion $h: U \times [m] \rightarrow [m]$.

Initialisiere **inUse** mit 0. (Zählt **nicht-leere** Zellen.)

Initialisiere **load** mit 0. (Zählt Einträge.)

Initialisiere **bound** (z. B. mit $\alpha_0 m$, maximal $m-1$).

lookup(x):

Finde erstes l in der Folge $h(x, 0), h(x, 1), h(x, 2), \dots$ mit

1. Fall: $T[l].\text{status} = \text{leer}$: **return** „*not found*“;

oder

2. Fall: ($T[l].\text{status} = \text{voll}$ und $T[l].\text{key} = x$): **return** $T[l].\text{data}$.

delete(x):

Finde erstes l in der Folge $h(x, 0), h(x, 1), h(x, 2), \dots$ mit

1. Fall: $T[l].\text{status} = \text{leer}$: tue nichts; // evtl. Warnung: *not found*

oder

2. Fall: ($T[l].\text{status} = \text{voll}$ und $T[l].\text{key} = x$): $T[l] \leftarrow (-, -, \text{gelöscht})$;
load--.

insert(x, r):

Finde erstes l in der Folge $h(x, 0), h(x, 1), h(x, 2), \dots$

mit $T[l].\text{status} = \text{leer}$ oder ($T[l].\text{status} = \text{voll}$ und $T[l].\text{key} = x$);

Merke dabei das erste l' in dieser Folge mit $T[l'].\text{status} \in \{\text{gelöscht}, \text{leer}\}$;

1. Fall: $T[l].\text{status} = \text{voll}$: $T[l].\text{data} \leftarrow r$; // Update

2. Fall: $T[l'].\text{status} = \text{gelöscht}$: $T[l'] \leftarrow (x, r, \text{voll})$; load++;
// überschreibe gelöschttes Feld

3. Fall: $T[l'].\text{status} = \text{leer}$: // $l = l'$, neue Zelle

if inUse + 1 > bound **then** „Überlaufbehandlung“

else inUse++; load++; $T[l] \leftarrow (x, r, \text{voll})$;

Überlaufbehandlung, Verdopplung:

Sinnvoll: **Überlauf** tritt ein, wenn die Anzahl der *voll*en und *gelöscht*en Zellen zusammen (in `inUse`) eine Schranke $\alpha_0 m$ (in `bound`) überschreitet.

(α_0 passend zum Sondierungsverfahren festlegen.)

Anzahl der *voll*en Zellen: `load`.

Falls $\text{load} + 1 > \alpha_1 m$ für (z. B.) $\alpha_1 = 0.8\alpha_0$, **verdopple** Tabellengröße, und trage Schlüssel aus T **neu** in die größere Tabelle ein. Setze `inUse` \leftarrow `load`.

Sonst: Trage alle Schlüssel **neu** in die **alte Tabelle** T ein, eliminiere alle „*gelöscht*“-Markierungen, wie folgt . . .

Trick für Umorganisieren: (Ohne zusätzlichen Platz.)

1) $\text{inUse} \leftarrow \text{load}$

2) Für $i = 0, 1, \dots, m - 1$ tue folgendes:

Wenn $T[i].\text{status} = \text{voll}$, setze $T[i].\text{status} \leftarrow \text{alt}$.

Wenn $T[i].\text{status} = \text{gelöscht}$, setze $T[i].\text{status} \leftarrow \text{leer}$.

3) Für $i = 0, 1, \dots, m - 1$ tue folgendes: // checke jeden Platz

Wenn $T[i].\text{status} = \text{alt}$, dann

setze $(x, r) \leftarrow (T[i].\text{key}, T[i].\text{data})$;

setze $T[i].\text{status} \leftarrow \text{leer}$;

insert* (x, r) .

Dabei ist **insert*** (x, r) folgende **rekursive** Variante von **insert** (x, r) :

Finde erstes l in der Folge $h(x, j)$, $j = 0, 1, 2, \dots$ mit $T[l].\text{status} \in \{\text{leer}, \text{alt}\}$;

Falls $T[l].\text{status} = \text{leer}$: $T[l] \leftarrow (x, r, \text{voll})$.

Falls $T[l].\text{status} = \text{alt}$: $(y, s) \leftarrow (T[l].\text{key}, T[l].\text{data})$;

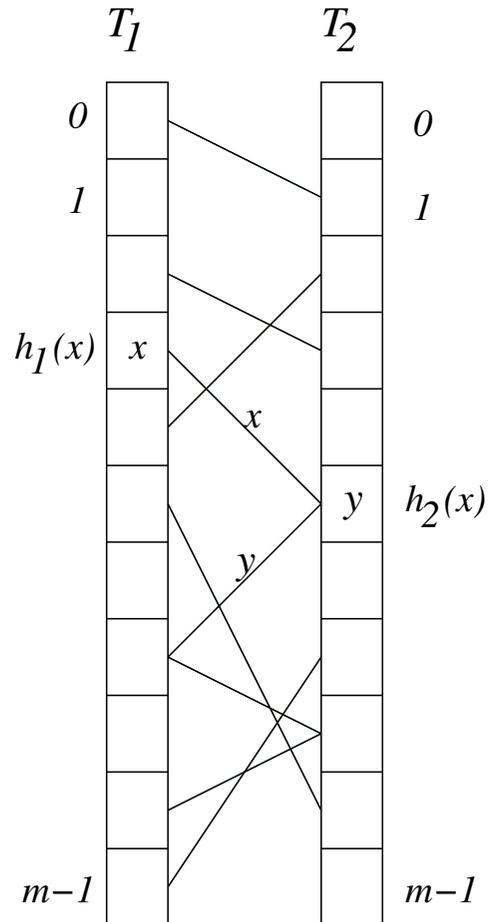
$T[l] \leftarrow (x, r, \text{voll})$;

insert* (y, s) .

Vorlesungsvideo:

Cuckoo Hashing

5.5 Cuckoo Hashing [Pagh/Rodler 2001/04]



Implementierung eines dynamisches Wörterbuchs
Zwei Tabellen T_1, T_2 , jeweils Größe m

Zwei Hashfunktionen h_1 und h_2

(*) $x \in S$ sitzt in $T_1[h_1(x)]$ oder in $T_2[h_2(x)]$.

⇒ **Konstante Suchzeit** garantiert.

Auch $delete(x)$ in $O(1)$ Zeit.

Cuckoo-Hashing

Definition 5.5.1

h_1, h_2 **passen zu** S , falls die Schlüssel aus S gemäß der Regel (*) gespeichert werden können, d. h. man kann S in S_1 und S_2 partitionieren, so dass h_1 auf S_1 und h_2 auf S_2 injektiv ist.

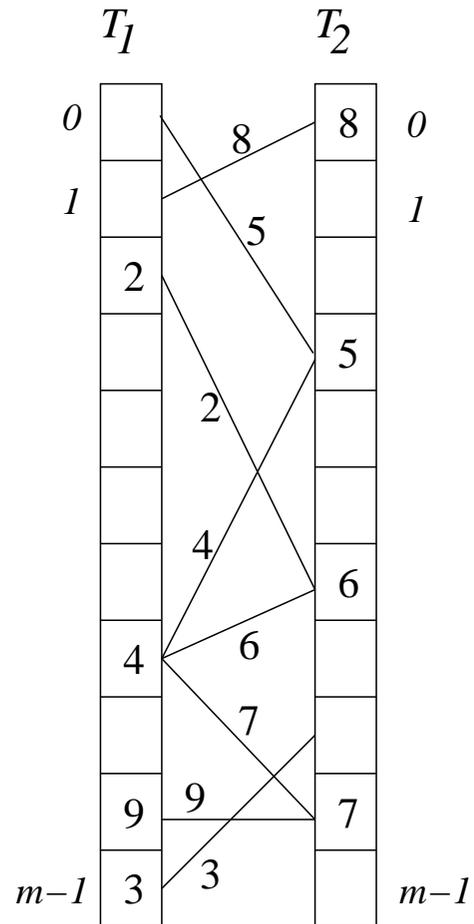
Das Verfahren heißt **“Kuckucks-Hashing”** wegen seiner Einfügeprozedur:

Schlüssel x , der eingefügt werden soll, kann Schlüssel y , der in $T_1[h_1(x)]$ oder $T_2[h_2(x)]$ („im Nest“) sitzt,

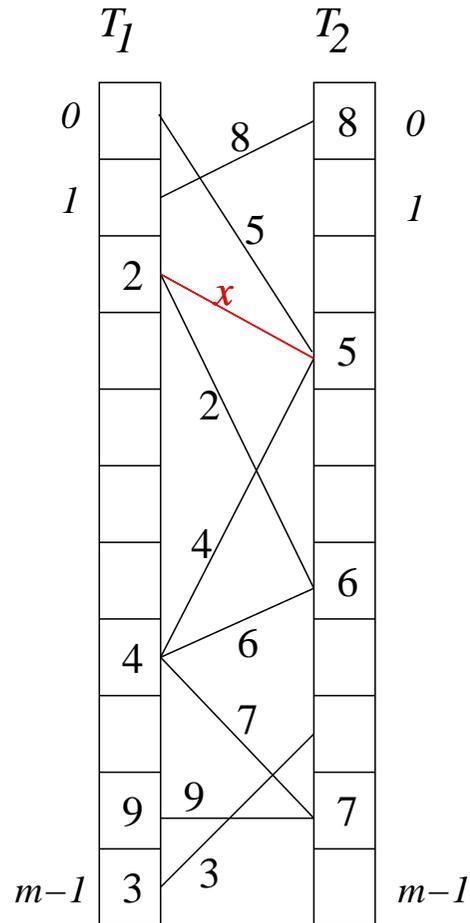
hinauswerfen (verdrängen).

Der verdrängte Schlüssel geht zu seinem alternativen Platz.

Cuckoo-Hashing

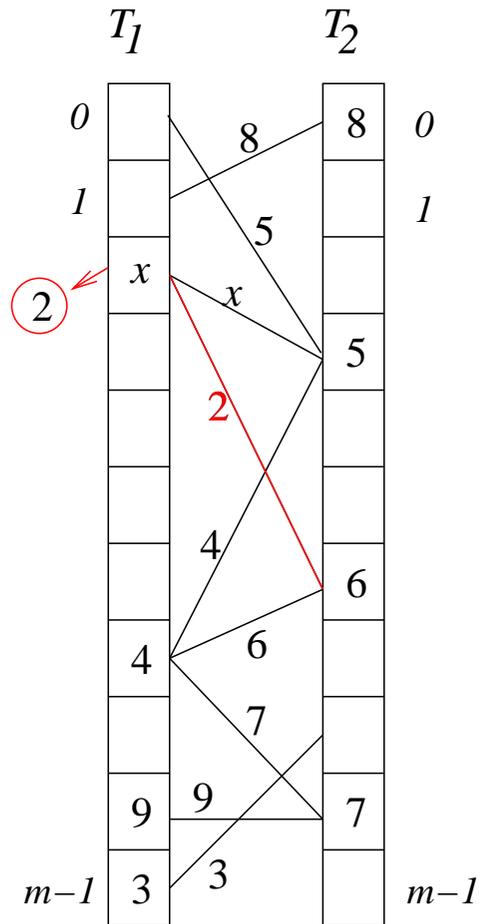


Cuckoo-Hashing



Füge x ein.
Versuche $T_1[h_1(x)]$.
Besetzt!

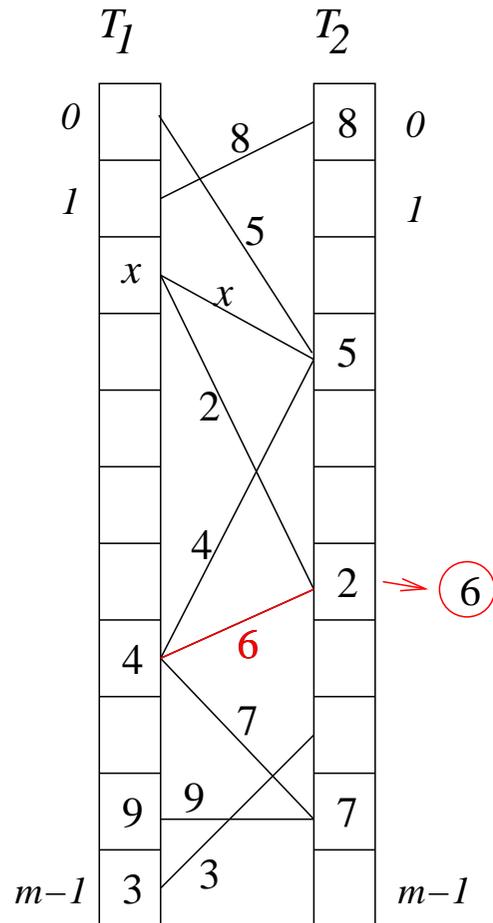
Cuckoo-Hashing



Schlüssel x in T_1 einfügen.

Schlüssel 2 aus T_1 hinauswerfen,
soll nach T_2 gehen.

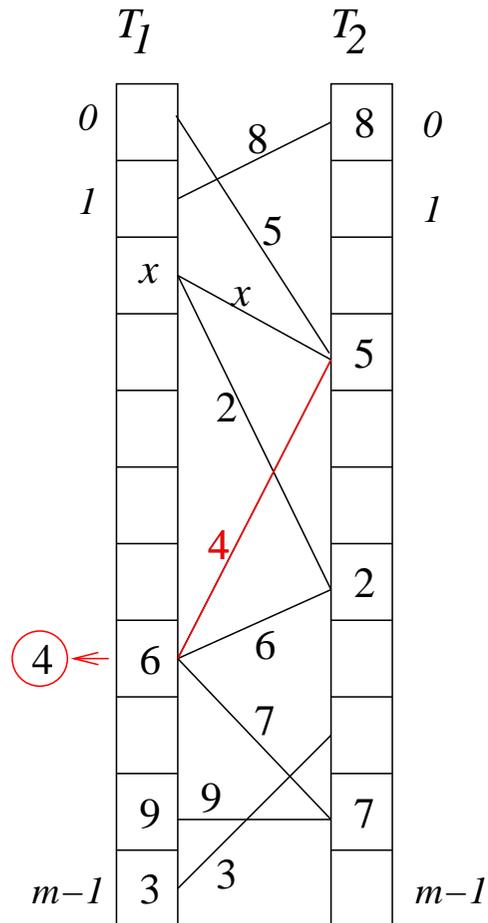
Cuckoo-Hashing



Schlüssel 2 in T_2 einfügen.

Schlüssel 6 aus T_2 hinauswerfen,
soll nach T_1 gehen.

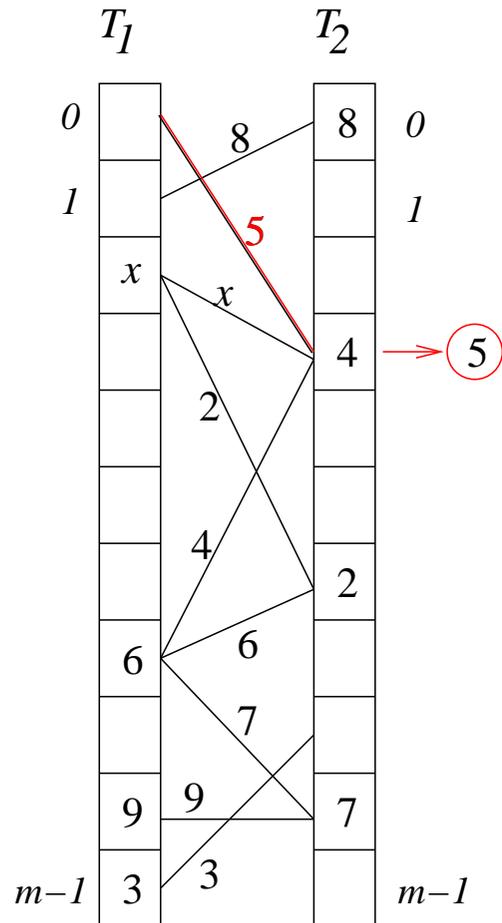
Cuckoo-Hashing



Schlüssel 6 in T_1 einfügen.

Schlüssel 4 aus T_1 hinauswerfen,
soll nach T_2 gehen.

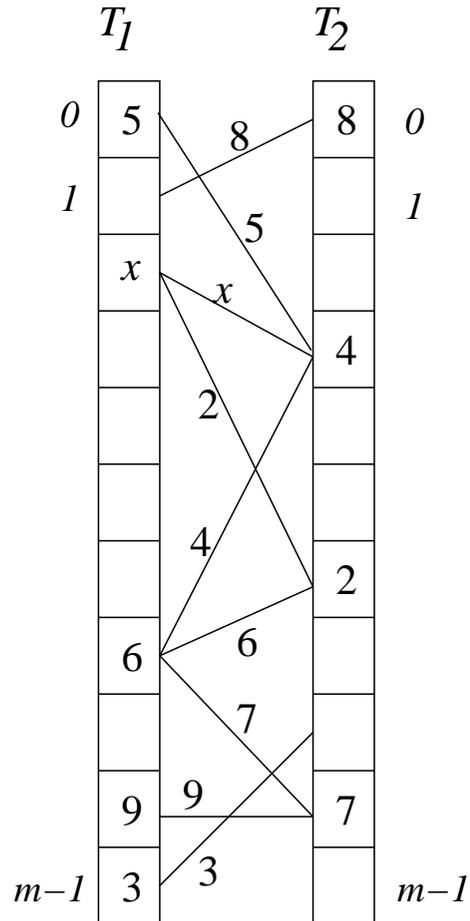
Cuckoo-Hashing



Schlüssel 4 in T_2 einfügen.

Schlüssel 5 aus T_2 hinauswerfen,
soll nach T_1 gehen.

Cuckoo-Hashing



Schlüssel 5 in T_1 einfügen.

Funktioniert, weil $T_1[h_1(5)]$ leer ist.

Fertig!

Cuckoo-Hashing: Teil-Analyse

Grundfrage: Wann funktioniert das überhaupt?

Hier zunächst: Wie lange dauert eine Einfügung?

Wie viele Verdrängungsschritte gibt es bei Einfügung von x ?

Wir beobachten: Wenn die Einfügung von x zur Verdrängung von y führt, dann gibt es eine Folge von Schlüsseln x_1, \dots, x_k mit:

x verdrängt x_1 , x_1 verdrängt x_2 , \dots , x_k verdrängt y .

Damit das passieren kann, muss es eine einfache „ $x..y$ -Kette“ $x, x_1, x_2, \dots, x_k, y$ aus *verschiedenen* Schlüsseln geben, die über die Hashfunktionen eine Verbindung herstellt:

$$h_1(x) = h_1(x_1),$$

$$h_2(x_1) = h_2(x_2)$$

$$h_1(x_2) = h_1(x_3)$$

$$\vdots$$

$$h_1(x_\ell) = h_1(y).$$

(Auch möglich: Die Folge beginnt mit $h_2(x) = h_2(x_1)$ oder sie endet mit $h_2(x_\ell) = h_2(y)$.)

Wir sammeln alle in Frage kommenden y in eine (Zufalls-)Menge

$$\mathbf{B}_x := \{y \in S \mid \text{es gibt eine (einfache) } x \dots y\text{-Kette}\}.$$

Wir wollen die erwartete Größe $\mathbf{E}(|B_x|)$ abschätzen.

Dazu schreiben wir $|B_x| = \sum_{y \in S} X_y$ mit

$$\mathbf{X}_y := \begin{cases} 1 & \text{wenn } y \in B_x \\ 0 & \text{wenn } y \notin B_x. \end{cases}$$

Dann gilt (Linearität des Erwartungswertes):

$$\mathbf{E}(|B_x|) = \sum_{y \in S} \mathbf{E}(X_y) = \sum_{y \in S} \mathbf{Pr}(\text{es gibt } x \dots y\text{-Kette}).$$

Wir schätzen $\mathbf{E}(X_y)$ mit Hilfe der „Vereinigungsschranke“ $\mathbf{Pr}(A_1 \cup \dots \cup A_\ell) \leq \mathbf{Pr}(A_1) + \dots + \mathbf{Pr}(A_\ell)$ ab:

$$\begin{aligned}\mathbf{E}(X_y) &= \mathbf{Pr}(\exists k \geq 1: \text{es gibt einfache } x \dots y\text{-Kette der Länge } k) \\ &\leq \sum_{k \geq 1} \sum_{x_1, \dots, x_k \in S} \mathbf{Pr}(x, x_1, \dots, x_k, y \text{ ist } x \dots y\text{-Kette}) \\ &\leq \sum_{k \geq 1} 4n^k \frac{1}{m^{k+1}}\end{aligned}$$

($k + 1$ Hashwerte sind gleich; Faktor 4 wg. h_1/h_2 -Auswahl)

$$\dots = \frac{4}{m} \cdot \sum_{k \geq 1} \left(\frac{n}{m}\right)^k \leq \frac{4}{m} \cdot \sum_{k \geq 0} (1 - \beta)^k = \frac{4}{m} \cdot \frac{1}{\beta},$$

falls $n/m \leq 1 - \beta$. Damit: $\mathbf{E}(|B_x|) \leq 4n/(m\beta) \leq \frac{4(1-\beta)}{\beta} < 4/\beta$.

Bei einer erfolgreichen Einfügung wird kein Schlüssel mehr als zweimal verdrängt.

Wir erhalten also

$$\mathbf{E}(\text{Einfügezeit}) = O(\mathbf{E}(|B_x|)) = O(1/\beta).$$

Leider ist die Analyse nicht ganz vollständig. Man muss noch die Wahrscheinlichkeit abschätzen, dass h_1 und h_2 nicht zu S passen. Diese ergibt sich (mit etwas komplizierteren Überlegungen) zu $\Theta(1/n)$.

Damit kann man dann die Analyse vervollständigen: Erwartete Einfügezeit ist $O(1)$, selbst wenn man den mit Wahrscheinlichkeit $O(1/n)$ nötigen Neuaufbau mit in Rechnung stellt.

Literatur: Rasmus Pagh, Cuckoo Hashing for Undergraduates, March 27, 2006, <https://www.itu.dk/people/pagh/papers/cuckoo-undergrad.pdf>

Cuckoo-Hashing: Analyse

Beobachtung 1 (trivial):

Wiederholte Anwendung der Einfügeprozedur platziert alle Schlüssel aus S in T_1, T_2

⇒

h_1, h_2 passen zu S .

D.h.: **Wenn** h_1, h_2 **nicht** zu S passen, **dann** muss die Einfügung eines Schlüssels **fehlschlagen**.

Wie sieht ein Fehlschlag aus?

Cuckoo-Hashing: Analyse

Wann gehen die Einfügungen gut?

Wie lange dauert eine Einfügung?

Beobachtung 1 (trivial):

Wiederholte Anwendung der Einfügeprozedur platziert alle Schlüssel aus S in T_1, T_2

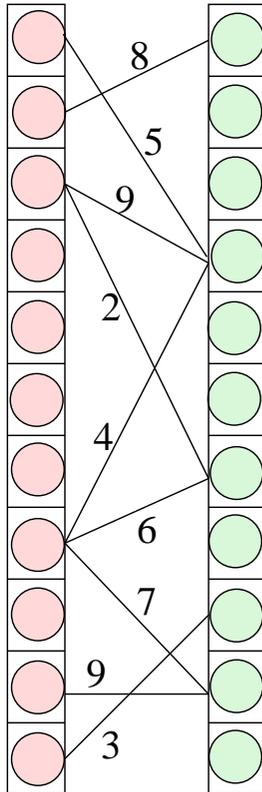
⇒

h_1, h_2 passen zu S .

D.h.: **Wenn** h_1, h_2 **nicht** zu S passen, **dann** muss die Einfügung eines Schlüssels **fehlschlagen**.

Wie sieht ein Fehlschlag aus?

Cuckoo-Hashing: Graph



Bipartiter

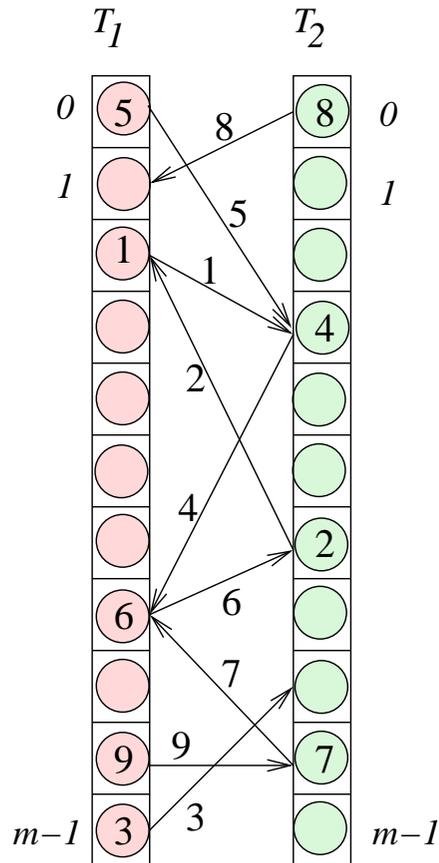
Cuckoo-Graph

$G(S, h_1, h_2)$

$$V = W = [m]$$

$$E = \{(h_1(x), h_2(x)) \mid x \in S\}$$

Cuckoo-Hashing: Graph

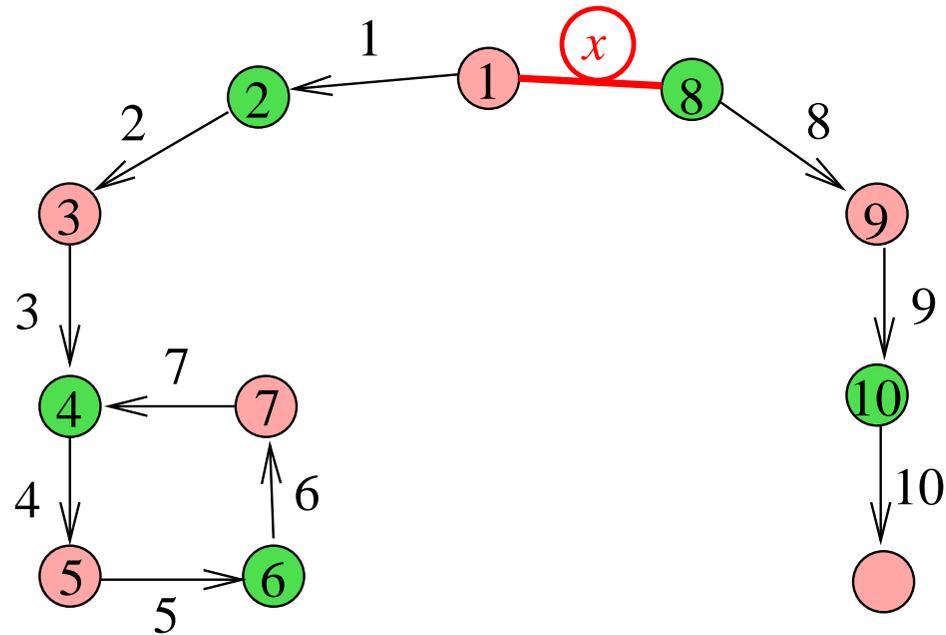


h_1, h_2 passen zu S

\Leftrightarrow

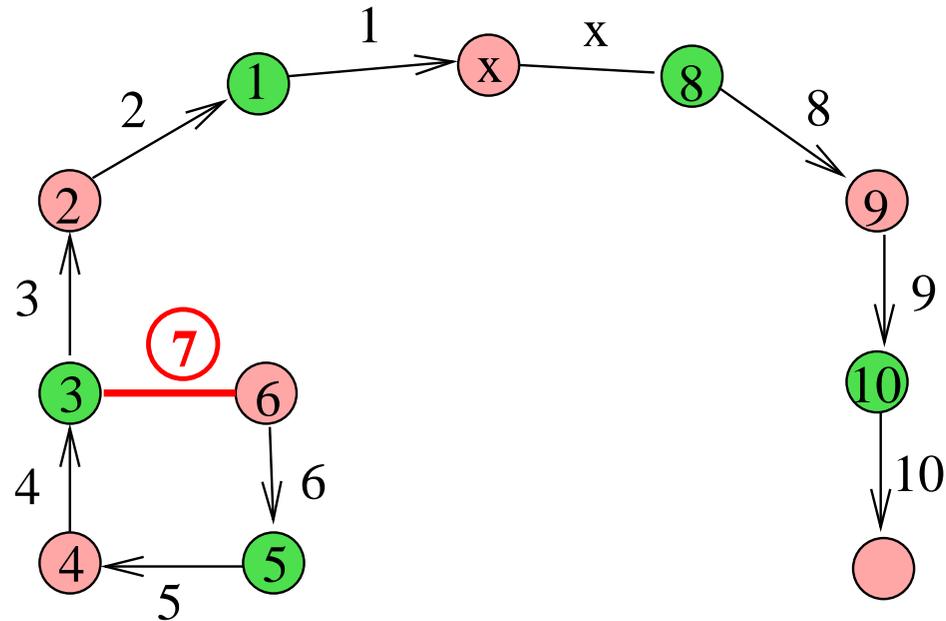
Man kann die Kanten
von $G(S, h_1, h_2)$
so richten,
dass jeder Knoten
Ausgrad ≤ 1 hat.

Cuckoo-Hashing: Fehlschlag?



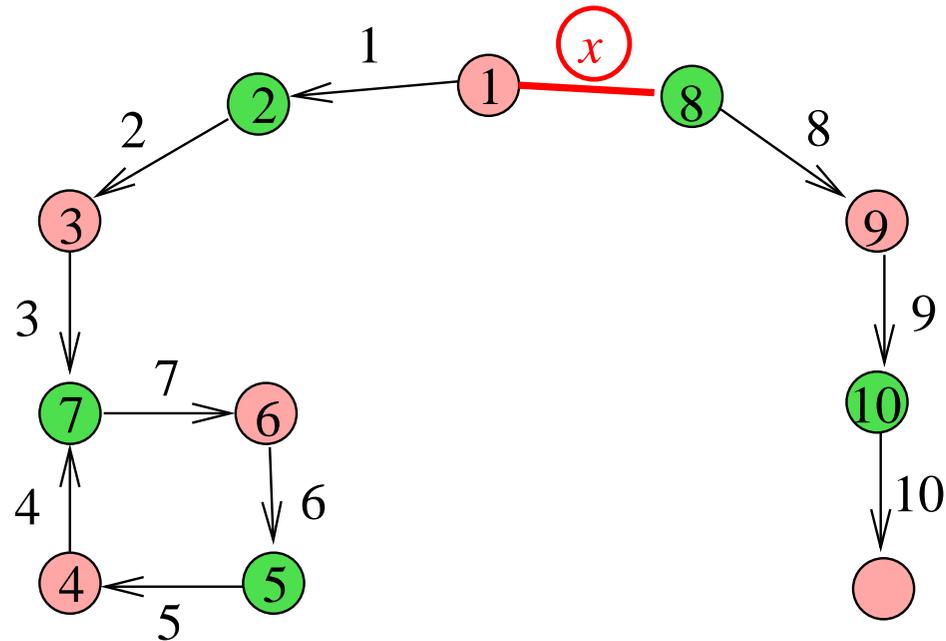
Neuer Schlüssel x .

Cuckoo-Hashing: Fehlschlag?



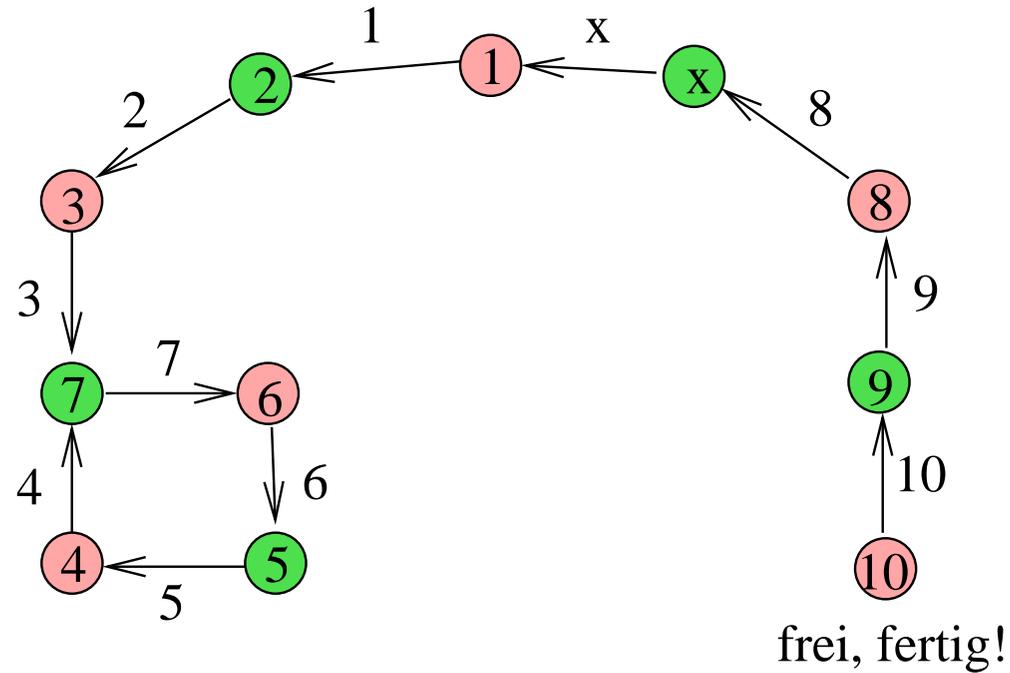
Nach 6 Runden. Schlüssel 7 ist draußen.

Cuckoo-Hashing: Fehlschlag?

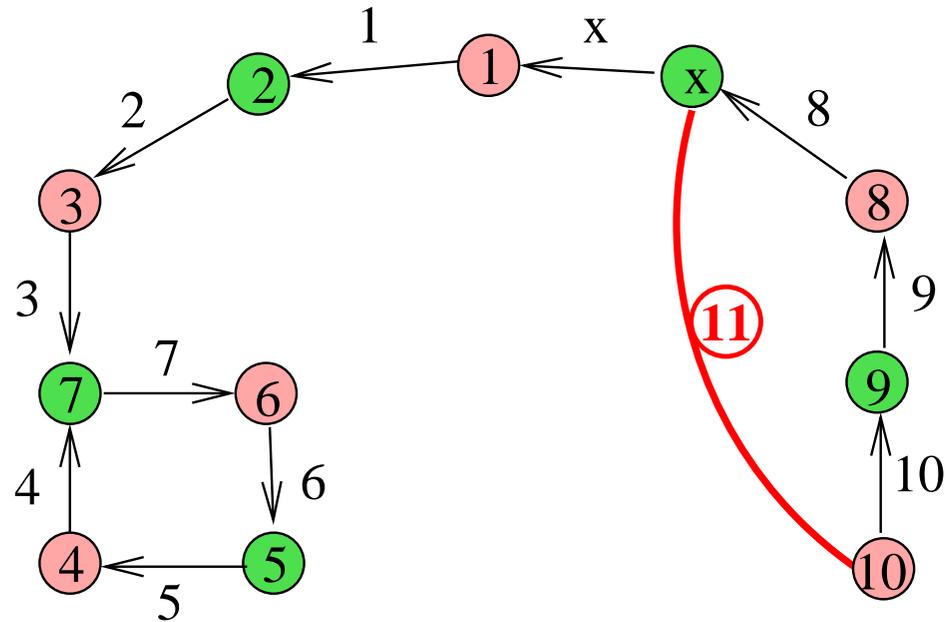


Nach weiteren 4 Runden.

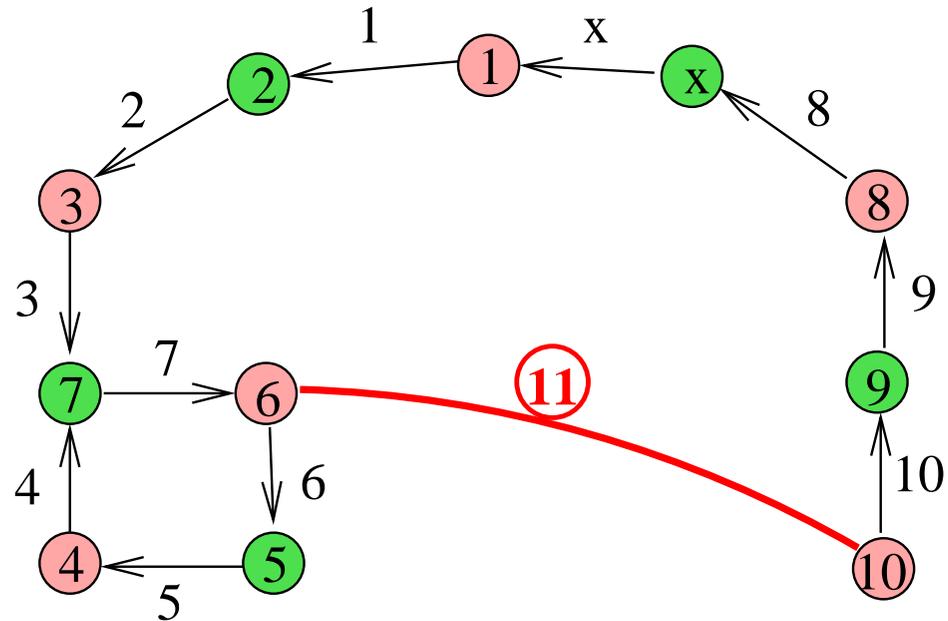
Cuckoo-Hashing: Fehlschlag?



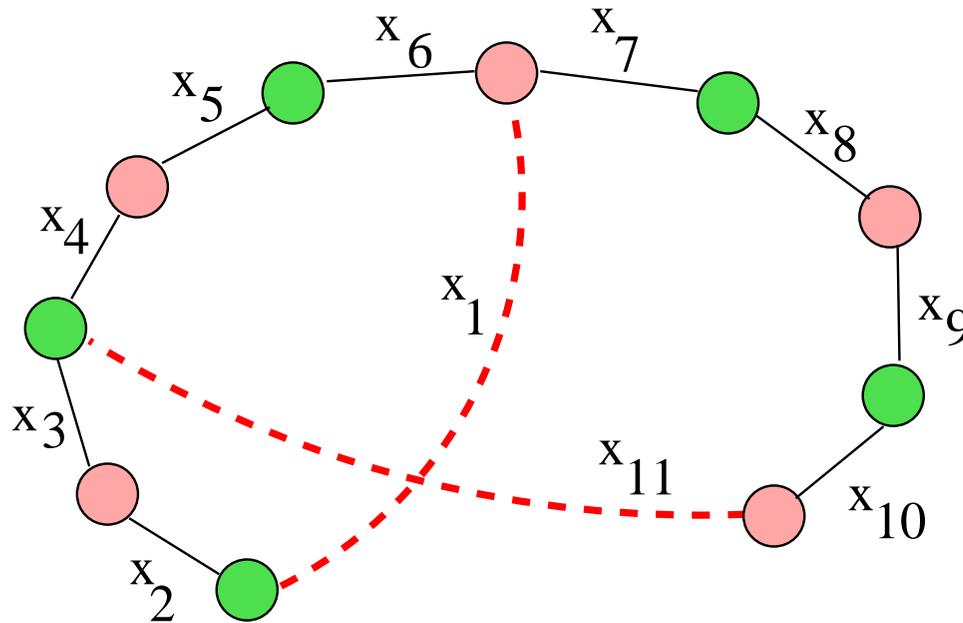
Cuckoo-Hashing: Fehlschlag!



Cuckoo-Hashing: Fehlschlag!

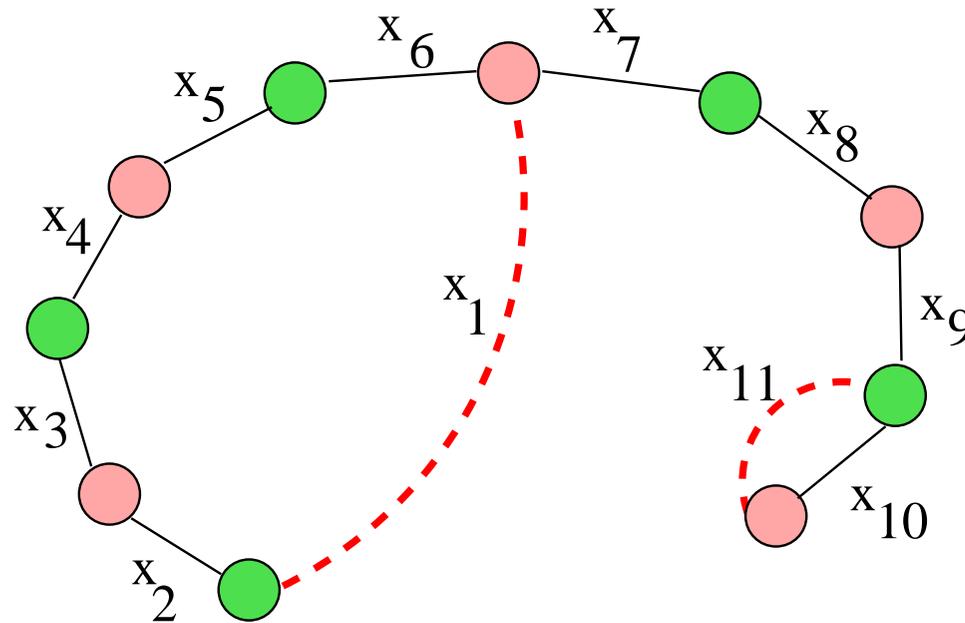


Cuckoo-Hashing: Fehlschlag



„Sperrstruktur“

Cuckoo-Hashing: Fehlschlag



„Sperrstruktur“

Cuckoo-Hashing: Analyse

Beobachtung 2: Aus dem Ablauf der Einfügung folgt:

Wenn h_1, h_2 **nicht** zu S passen, **dann** existiert in $G(S, h_1, h_2)$ eine **Sperrstruktur**, d. h. es gibt eine Folge von Schlüsseln x_1, \dots, x_k in S , deren Hashwerte wie folgt kollidieren (z. B., falls k ungerade):

$$h_2(x_1) = h_2(x_2),$$

$$h_1(x_2) = h_1(x_3),$$

$$h_2(x_3) = h_2(x_4),$$

$$h_1(x_4) = h_1(x_5),$$

⋮

⋮

$$h_2(x_{k-2}) = h_2(x_{k-1}),$$

$$h_1(x_{k-1}) = h_1(x_k),$$

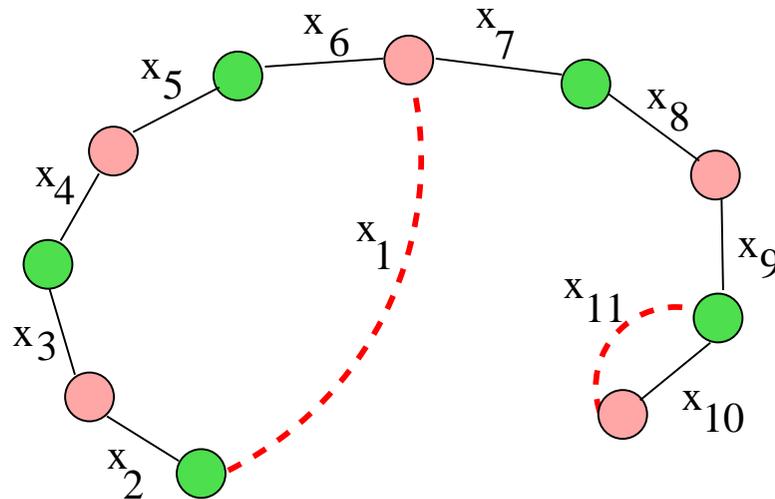
$$h_1(x_1) \in \{h_1(x_3), \dots, h_1(x_k)\},$$

$$h_2(x_k) \in \{h_2(x_2), \dots, h_2(x_{k-1})\}.$$

Cuckoo-Hashing: Analyse

Beobachtung 2': Sogar Äquivalenz!

Wenn in $G(S, h_1, h_2)$ eine **Sperrstruktur** existiert,
dann passen h_1, h_2 nicht zu S .



Mehr Kanten/Schlüssel als Knoten/Tabellenplätze!

Cuckoo-Hashing: Analyse

Sei $n/m \leq 1 - \beta$. Dann gilt:

$\Pr(h_1, h_2 \text{ passen nicht zu } S)$

$$\begin{aligned} &\leq \sum_{3 \leq k \leq n} n^k \cdot 2 \cdot \left(\frac{1}{m}\right)^{k-1} \cdot \left(\frac{k}{2m}\right)^2 \\ &< \frac{1}{2m} \cdot \sum_{k \geq 3} k^2 \cdot \left(\frac{n}{m}\right)^k \\ &\leq \frac{1}{2m} \cdot \sum_{k \geq 3} k^2 \cdot (1 - \beta)^k = O\left(\frac{1}{\beta^3 \cdot m}\right). \end{aligned}$$

Cuckoo-Hashing: Analyse

(Es gibt $\leq n^k$ Folgen (x_1, \dots, x_k) von Schlüsseln.

Die erste Kante kann einen h_1 - oder einen h_2 -Endpunkt haben.

Dass die Hashwerte entlang des Weges übereinstimmen, hat Wahrscheinlichkeit $1/m^{k-1}$.

Die Anzahl der möglichen Trefferknoten für jede der beiden „zurückschlagenden“ Kanten ist $\leq k/2$.)

Cuckoo-Hashing: Analyse

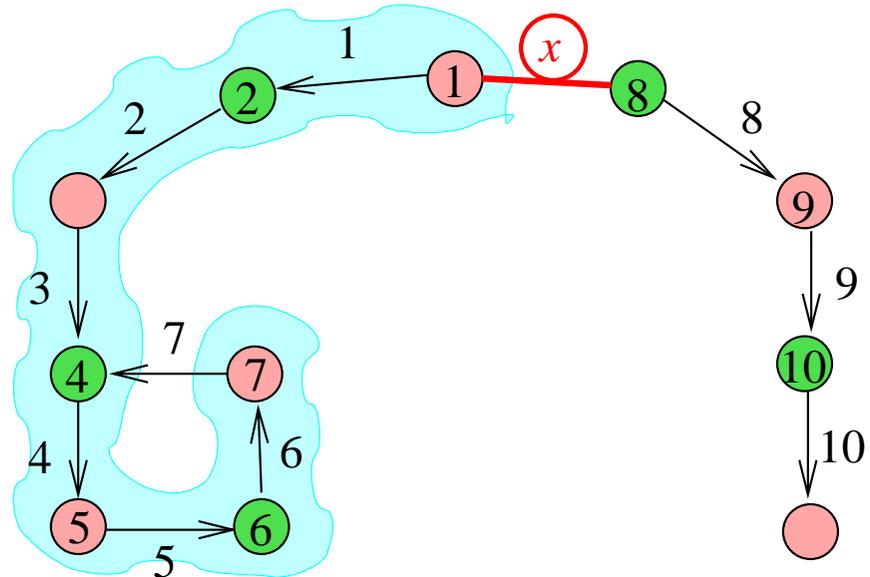
Benötige: Mechanismus, um Endlosschleifen (Einfügung, die Sperrstruktur erzeugt) abzubrechen.

Lasse Einfügezyklus für **maximal** $C \log n$ Schritte laufen, C eine Konstante.

Danach: „Rehash“ (neue Hashfunktion, Neuaufbau).

Will aber keine erfolgreiche Einfügung verlieren, nur weil sie lange dauert!

Cuckoo-Hashing: Analyse



Beobachtung 3:

Wenn die **erfolgreiche** Einfügung von x mindestens t Verdrängungen durchführt, dann gibt es in $G(S, h_1, h_2)$ einen einfachen Weg der Länge $\geq \lceil (t - 1)/3 \rceil$, der an Knoten $h_1(x)$ oder $h_2(x)$ beginnt.

Cuckoo-Hashing: Analyse

Pr($\geq t$ Verdrängungsschritte)

$$\leq 2 \cdot n^{\lceil (t-1)/3 \rceil} \left(\frac{1}{m} \right)^{\lceil (t-1)/3 \rceil}$$

$$\leq 2 \cdot (1 - \beta)^{(t-1)/3}$$

$$\leq 2^{c-t/d}, \text{ für Konstante } c, d.$$

\Rightarrow **Pr**($\geq C \log n$ Verdrängungsschritte für x) = $O(1/m^3)$
für Konstante $C \geq \frac{3 \ln 2}{\beta}$ [$\geq 3 / \log(1/(1 - \beta))$].

\Rightarrow **Pr**($\exists x \in S$: $\geq C \log n$ Verdrängungsschritte für x) = $O(1/m^2)$.

Cuckoo-Hashing: Analyse

Einfügung von x kostet:

$E(\text{Anzahl Verdrängungsschritte})$

$$\begin{aligned} &\stackrel{*}{\leq} \sum_{t \geq 1} \Pr(\geq t \text{ Verdrängungsschritte}) \\ &\leq \sum_{t \geq 1} 2^{c-t/d} \\ &= O(1). \end{aligned}$$

* Standardformel für Erwartungswerte.

Cuckoo-Hashing: Zusammenfassung

Mitteilung 5.5.2 Vor.: **(UF*)** Hashfunktionen verteilen Schlüssel **rein zufällig**. Wenn man Cuckoo-Hashing mit n Schlüsseln in zwei Tabellen mit je m Plätzen durchführt, wobei **$n/m \leq 1 - \beta$ (essenziell!)**, dann passen h_1, h_2 mit Wahrscheinlichkeit $1 - O(1/m)$ zur Schlüsselmenge S .

Falls h_1, h_2 passen, ist die erwartete Einfügezeit $O(1/\beta)$.

Bei Auftreten eines Fehlers (z. B. Einfügung nach $10 \log n$ Runden nicht beendet): „Rehash“ (neue Hashfunktionen wählen) mit erwarteten Kosten $O(n)$. „Amortisierte“ erwartete Einfügezeit: $O(1)$.

Literatur: [Rasmus Pagh, Cuckoo Hashing for Undergraduates],
<https://www.itu.dk/people/pagh/papers/cuckoo-undergrad.pdf>

Nachteil: Auslastung der Tabelle **unter 50%**.

Abhilfe: Mehr als 2 Hashfunktionen oder größere Behälter. (Details in Spezialvorlesungen.)

Die verfeinerten Verfahren sind auch **praktisch** sehr effizient (auch mit Tabellenhashing).

Multiplikative/lineare Hashfunktionen nicht benutzen!