Algorithmen und Datenstrukturen

Vorlesung 5

Hash Tabellen 2: Hashfunktionen, Universelles Hashing, Rehash, Cuckoo Hashing

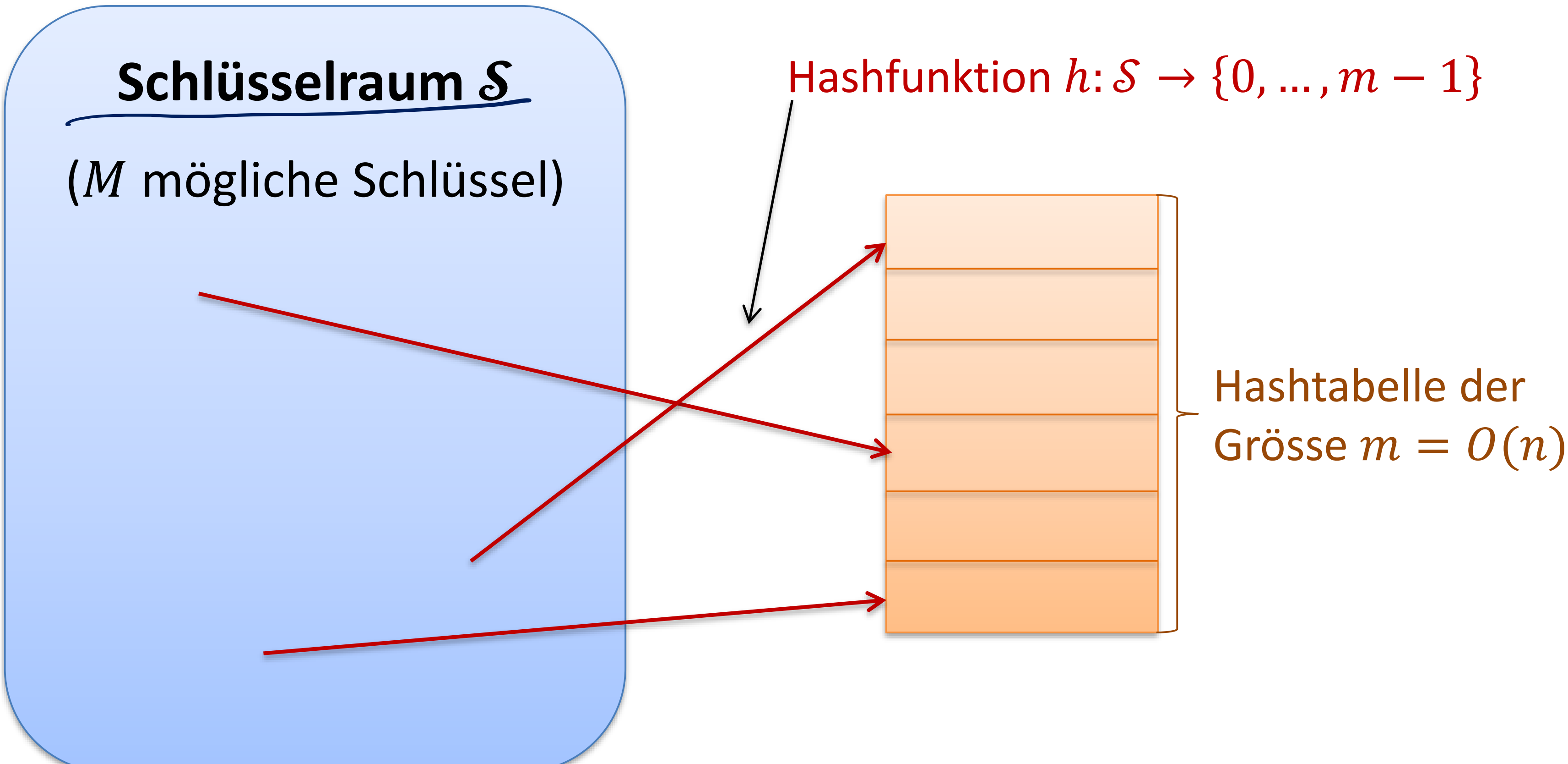
Fabian Kuhn
Algorithmen und Komplexität

UNI FREIBURG

Hashtabellen

Implementiert einen Dictionary

- Verwalten eine Menge an (Schlüssel, Wert)-Paaren
- Hauptoperationen: insert, find, delete



Hashtabellen

Wir haben bisher gesehen:

effiziente Methode, um einen Dictionary zu implementieren

- Alle Operationen haben typischerweise $O(1)$ Laufzeit
  - Falls die Hashfunktionen genug zufällig sind und in $O(1)$ Zeit ausgewertet werden können.
  - Die Worst-Case Laufzeit ist etwas höher, in jeder Anwendung von Hashfunktionen wird es ein paar teurere Operationen dabei haben.

Wir werden uns nun anschauen:

- Wie wählt man eine gute Hashfunktion?
- Was macht man, wenn die Hashtabelle zu klein wird?
- Man kann Hashing so implementieren, dass find immer in $O(1)$ Zeit implementiert werden kann.

Gute Hashfunktionen

Wie wählt man eine gute Hashfunktion?

Was sollte eine gute Hashfunktion erfüllen?

- Im Prinzip sollte sie die gleichen Eigenschaften wie eine zufällige Funktion haben:
  – Mapping ist uniform zufällig (alle Hashwerte kommen gleich oft vor)
  – Mapping von verschiedenen Schlüsseln ist unabhängig (nicht klar, was das bei einer deterministischen Funktion genau heissen soll)
- Man kann diese Bedingungen meistens nicht überprüfen
- Falls man etwas über die Verteilung der Schlüssel weiss, kann man das allenfalls ausnützen
- Es gibt aber zum Glück einfache Heuristiken, welche in der Praxis gut funktionieren

Divisionsmethode

Grösse der Hashtabelle

Wähle Hashfunktion als

$$\boldsymbol{h(x) = x \ \mathbf{mod}\ m}$$

- Alle Werte zwischen 0 und $m - 1$ kommen gleich oft vor
  - so gut, wie das möglich ist

Vorteile:

- Sehr einfache Funktion
- Nur eine Division → kann man schnell berechnen
- Funktioniert oft recht gut, solange man m geschickt wählt...

Bemerkungen:

- Falls die Schlüssel keine ganzen Zahlen sind, kann man den Bitstring als ganze Zahl interpretieren
- Aufeinanderfolgende Schlüssel werden auf aufeinanderfolgende Hashwerte abgebildet

Divisionsmethode

Wähle Hashfunktion als

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

$10100110101\underbrace{01110101}$

Wahl des Divisors m

- Man könnte $h(x)$ besonders schnell berechnen falls $m = 2^k$
- Das ist aber keine gute Wahl, da man dann einfach die letzten k Bits als Hashwert bekommt!
  - Der Hashwert sollte von allen Bits abhängen
- Am besten wählt man m als Primzahl
- Eine Primzahl m, so dass $m = 2^k - 1$ ist auch keine gute Idee
- Am besten: Primzahl m, welche nicht nahe bei einer 2er-Potenz ist

Multiplikationsmethode

Wähle Hashfunktion als

$$h(x) = \lfloor m \cdot (\overbrace{\underbrace{Ax - \lfloor Ax \rfloor}_{\in [0,1)}}^{0 \le Ax - \lfloor Ax \rfloor < 1}) \rfloor$$

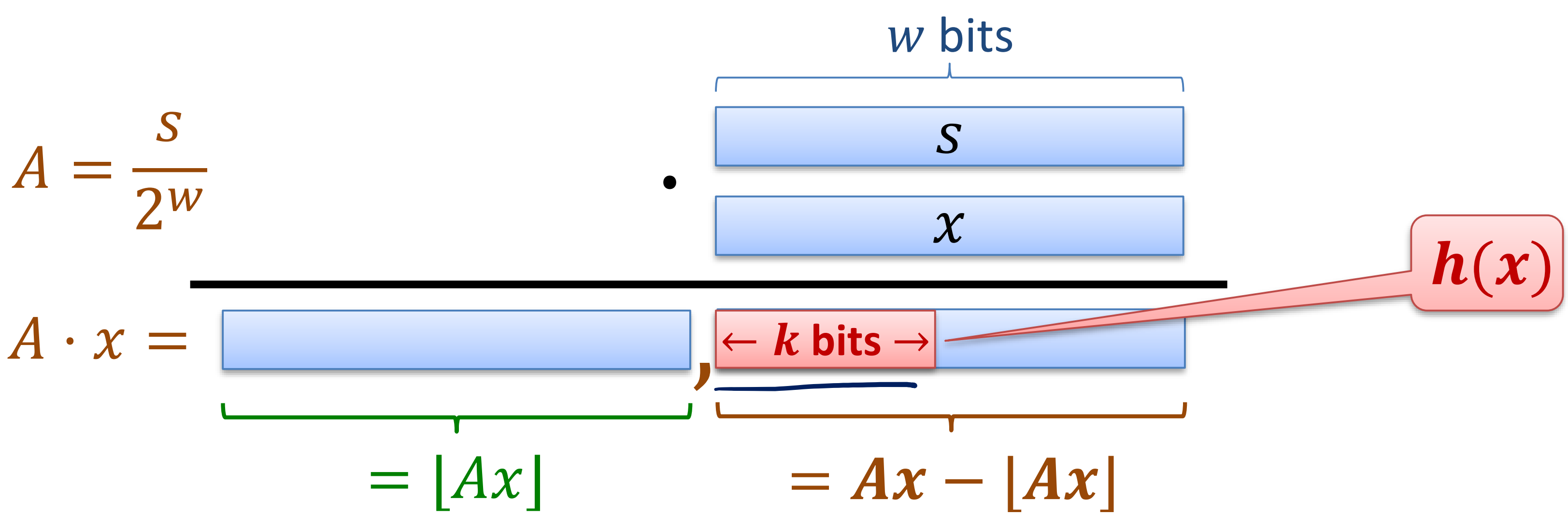
$$A \cdot x = \frac{s \cdot x}{2^w}$$

- A ist eine Konstante zwischen 0 und 1

$$A = \frac{s}{2^w}$$

Bemerkungen

- Hier kann man $m = 2^k$ wählen (für Integer k)
- Falls Integers von 0 bis $2^w - 1$ gehen, wählt man typischerweise einen Integer $s \in \{1, \dots, 2^w - 1\}$ und $A = s \cdot 2^{-w}$



Multiplikationsmethode

Wähle Hashfunktion als

$$h(x) = \lfloor m \cdot (Ax - \lfloor Ax \rfloor) \rfloor$$

- A ist eine Konstante zwischen 0 und 1

Bemerkungen

- Hier kann man $m = 2^k$ wählen (für Integer k)
- Falls Integers von 0 bis $2^w - 1$ gehen, wählt man typischerweise einen Integer $s \in \{1, \dots, 2^w - 1\}$ und $A = s \cdot 2^{-w}$
  - Grundsätzlich funktioniert jedes A, in [Knuth; The Art of Comp. Progr. Vol. 3] wird empfohlen, dass

$$A \approx \frac{\sqrt{5} - 1}{2} = 0.6180339887 \dots$$

Zufällige Hashfunktionen

Falls h zufällig aus allen möglichen Funktionen ausgewählt wird:

$$\forall x_1, x_2 : \Pr\big(h(x_1) = h(x_2)\big) = \frac{1}{m}$$

und viele weitere gute Eigenschaften …

Problem:

- eine solche Funktion kann nicht effizient repräsentiert und ausgewertet werden
  - Im Wesentlichen braucht man eine Tabelle mit allen möglichen Schlüsseln

Idee:

- Eine Funktion zufällig aus einem kleineren Bereich wählen
  - z.B. bei Multiplikationsmethode $h(x) = \lfloor m \cdot (Ax - \lfloor Ax \rfloor) \rfloor$ einfach den Parameter A zufällig wählen
- Nicht ganz so gut, wie eine uniform zufällige Funktion, aber wenn man's richtig macht, funktioniert die Idee → **universelles Hashing**

Universelles Hashing : Idee

Hashfunktionen: $h : \mathcal{S} \rightarrow \{0, \dots, m-1\}$

Schlüsselraum $\mathcal{S}$

$\mathcal{S} = \{0, \dots, M-1\}$

$\xrightarrow{h}$ Positionen $0, \dots, m-1$

Raum aller möglichen Hashfunktionen

Mögliche Hashfunktionen
(Anz. Funktionen: m^M)

Teilmenge
$\mathcal{H}$

Wähle $\mathcal{H}$ so, dass:

- $|\mathcal{H}|$ nicht allzu gross ist und Funktionen in $\mathcal{H}$ einfach zu implementieren sind
- Eine zufällige Funktion h aus $\mathcal{H}$ verhält sich ähnlich, wie eine zufällige Funktion
- Insb. bezügl. Kollisionswahrscheinlichkeit:

$$\forall x_1, x_2 : \Pr\big(h(x_1) = h(x_2)\big) \approx \frac{1}{m}$$

Universelles Hashing : Definition

$\mathcal{H}$

Hashfkt. h mit $h(x)=h(y)$

B_{xy}

Definition:

- Sei $\mathcal{S}$ die Menge der mögl. Schlüssel und m die Grösse der Hashtab.
- Sei $\mathcal{H}$ eine Menge von Hashfunktionen $\mathcal{S} \to \{0, \dots, m-1\}$

$|B_{xy}| \leq \frac{c}{m} \cdot |\mathcal{H}|$

Die Menge $\mathcal{H}$ heisst c-universell, falls $(c \geq 1)$

$$\forall x, y \in \mathcal{S} : x \neq y \Longrightarrow |\{h \in \mathcal{H} : h(x) = h(y)\}| \leq c \cdot \frac{|\mathcal{H}|}{m}.$$

- Mit anderen Worten, falls man h zufällig aus $\mathcal{H}$ wählt, dann gilt

$$\forall x, y \in S : x \neq y \Longrightarrow \Pr\big(h(x) = h(y)\big) \leq \frac{c}{m}$$

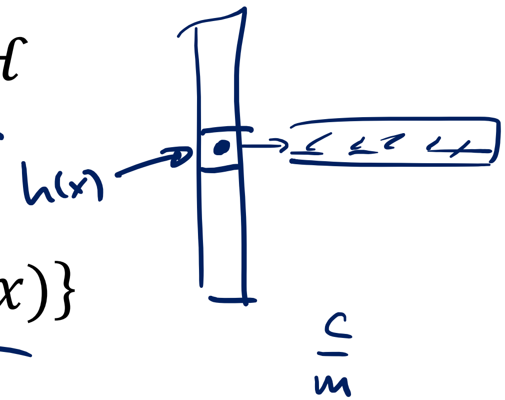
- **Bemerkung:**
  Die Menge $\mathcal{H}$ aller m^M möglichen Hashfunktionen ist 1-universell.

Universelles Hashing : Listenlängen

Theorem:

- Sei $\mathcal{H}$ eine c-universelle Menge von Hashfkt. $\mathcal{S} \to \{0, \dots, m-1\}$
- Sei $X \subset \mathcal{S}$ eine beliebige Menge von Schlüsseln
- Sei $h \in \mathcal{H}$ eine zufällig gewählte Fkt. aus $\mathcal{H}$
- Für ein gegebenes $x \in X$ sei

$$B_x \coloneqq \{y \in X : h(y) = h(x)\}$$



- Im Erwartungswert hat B_x Grösse $< 1 + c \cdot \frac{|X|}{m}$

Konsequenz:

- Im Erwartungswert sind alle Listen kurz!

Universelles Hashing : Beispiele I

Die Menge $\mathcal{H}$ heisst c-universell, falls

$$\forall x, y \in \mathcal{S} : x \neq y \Longrightarrow |\{h \in \mathcal{H} : h(x) = h(y)\}| \leq c \cdot \frac{|\mathcal{H}|}{m}.$$

Negatives Beispiel:

- Parametriesierte Variante der Divisionsmethode

$$\mathcal{H} = \{h : x \to a \cdot x \mod m \quad \text{for } a \in \{1, \dots, M-1\}\}$$

- Gegenbeispiel: wähle beliebiges x und wähle $y = x + m$
  - $h(x) = a \cdot x \mod m$
  - $h(y) = a \cdot (x + m) \mod m = (a \cdot x + a \cdot m) \mod m = a \cdot x \mod m$

Universelles Hashing : Beispiele II

Die Menge $\mathcal{H}$ heisst c-universell, falls

$$\forall x, y \in \mathcal{S} : x \neq y \Longrightarrow |\{h \in \mathcal{H} : h(x) = h(y)\}| \leq c \cdot \frac{|\mathcal{H}|}{m}.$$

$ax+b = ay+b+i\cdot m$

Positives Beispiel 1:

- m beliebig, p: Primzahl mit $p > M$

  $$\mathcal{H} = \{h : x \to ((a \cdot x + b) \bmod p) \bmod m \quad \text{for } a, b \in \mathcal{S}, a \neq 0\}$$

- Die Familie ist c-universell für $c \approx 1$, falls $p \approx M$
- Für x, y gilt $h(x) = h(y)$, falls für ein $i \in \mathbb{Z}$ gilt:

  $$(ax + b) \bmod p = (ay + b) \bmod p + i \cdot m$$

  $$\boxed{a \equiv i \cdot m \cdot (x - y)^{-1} \pmod{p}}$$

  gilt für höchstens $2 \cdot \left\lceil \frac{p-1}{m} \right\rceil + 1$ versch. i-Werte

- Für jedes x und y und jedes b gibt es also für jedes mögliche i nur einen Wert a, für welchen x und y kollidieren.

Universelles Hashing : Beispiele III

Die Menge $\mathcal{H}$ heisst c-universell, falls

$$\forall x, y \in \mathcal{S} : x \neq y \Longrightarrow |\{h \in \mathcal{H} : h(x) = h(y)\}| \leq c \cdot \frac{|\mathcal{H}|}{m}.$$

Positives Beispiel 2:

- m Primzahl, $k = \lceil \log_m M \rceil - 1$, Parameter $a \in \mathcal{S} = \{0, \dots, M-1\}$
- Betrachte Parameter a und Schlüssel x in Basis-m Darstellung:

$$a = a_0 + a_1 \cdot m + a_2 \cdot m^2 + \cdots + a_k \cdot m^k$$
$$x = x_0 + x_1 \cdot m + x_2 \cdot m^2 + \cdots + x_k \cdot m^k$$

$a_i, x_i \in \{0, \dots, m-1\}$

$$\mathcal{H} = \left\{ h : x \to \left(\sum_{i=0}^{k-1} a_i \cdot x_i \right) \bmod m \;\; \text{for } a_i \in \{0, \dots, m-1\} \right\}$$

- Die Menge $\mathcal{H}$ ist 1-universell

Universelles Hashing : Zusammenfassung

- Wenn man die Hashfunktion zufällig aus einer universellen Menge von Hashfunktionen wählt, ist die Kollisionswahrscheinlichkeit für zwei Schlüssel x und y gleich, wie bei einer zufälligen Fkt.
- Es gibt einfache und effiziente Konstruktionen von universellen Mengen von Hashfunktionen

Man kann noch weiter gehen:

- Paarweise unabhängige Mengen von Hashfunktionen

$$\forall x, y \in \mathcal{S}, \forall a, b \in \mathbb{Z}_m : \Pr(h(x) = a \ \land \ h(y) = b) = \frac{1}{m^2}$$

  – eine zufällige Funktion aus einer solchen Menge verhält sich für alle Paare x, y genau gleich wie eine zufällige Funktion (nicht nur bezügl. Kollisionen)

- k-fach unabhängige Mengen von Hashfunktionen
  – eine zufällige Funktion aus einer solchen Menge verhält sich für alle k verschiedenen Schlüssel genau gleich wie eine zufällige Funktion

Rehash

Erinnerung:

- Load einer Hashtabelle: $\alpha = n/m$

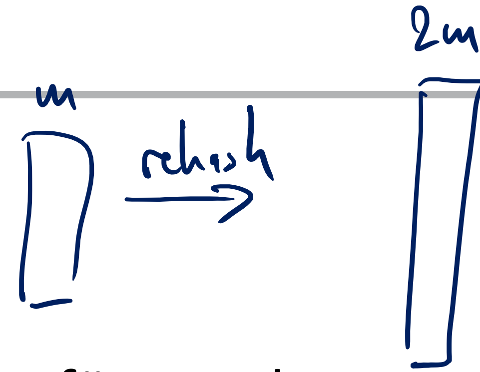
Was tun, wenn die Hashtabelle zu voll wird?

- Offene Adressierung:
  - $\alpha > 1$ nicht möglich, bei $\alpha \to 1$ sehr ineffizient
  - wenn man viel eingefügt und wieder gelöscht hat, wird es auch ineffizient (wegen der delete-Markierungen)
- Chaining: Komplexität wächst linear mit α

Was tun, wenn die gewählte Hashfunktion schlecht ist?

Rehash:
- Erstelle neue, grössere Hashtabelle, wähle neue Hashfunktion h'
- Füge alle Schlüssel/Werte neu ein

Kosten Rehash



Ein Rehash ist teuer!

Kosten (Zeit):

- $\Theta(m + n)$: linear in der Anzahl eingefügten Elemente und der Länge der alten Hashtabelle
  - typischerweise ist das einfach $\Theta(n)$

- **Wenn man es richtig macht, ist ein Rehash selten nötig:**
  - gute Hashfunktion (z.B. aus einer universellen Klasse)
  - gute Wahl der Tabellengrössen:

    bei jedem **Rehash** sollte die **Tabellengrösse** etwa **verdoppelt** werden

    alte Grösse $m \implies$ neue Grösse $\approx 2m$
  - Verdoppeln ergibt immer noch durchschnittlich konstante Zeit pro Hashtabellen-Operation

    → amortisierte Analyse

Kosten Rehash

Analyse Verdoppelungsstrategie

- Wir machen ein paar vereinfachende Annahmen:
  - Bis zu Load α_0 (z.B. $\alpha_0 = {}^1/_2$) kosten alle Hashtabellen-Operationen $\leq c$
  - Bei Load α_0 wird die Tabellengrösse verdoppelt:
    Alte Grösse m, neue Grösse $2m$, Kosten $\leq c \cdot m$
  - Am Anfang hat die Tabelle Grösse $m_0 \in O(1)$
  - Die Tabelle wird nie verkleinert…

- Wie gross sind die Kosten für das Rehashing, verglichen mit den Gesamtkosten für alle anderen Operationen?

Kosten Rehash

$$m_0 \xrightarrow{c m_0} 2m_0 \xrightarrow{2c m_0} 4m_0$$

Gesamtkosten

- Wir nehmen an, dass die Tabellengrösse $m = m_0 \cdot 2^k$ für $k \geq 1$ ist
  - d.h., bis jetzt haben wir $k \geq 1$ Rehash-Schritte gemacht
  - Bemerkung: Bei $k = 0$ sind die Rehash-Kosten 0.

$m_0 2^{k-1} \to m_0 \cdot 2^k$

$\frac{m}{2} \to m$

- Die Gesamt-Rehash-Kosten sind dann

$$\leq \sum_{i=0}^{k-1} c \cdot m_0 \cdot 2^i = c \cdot m_0 \cdot (2^k - 1) \leq c \cdot m$$

$\alpha_0 \cdot \frac{m}{2}$

- Gesamt-Kosten für die übrigen Operationen
  - Beim Rehash von Grösse ${}^{m}/_{2}$ auf m waren $\geq \alpha_0 \cdot {}^{m}/_{2}$ Einträge in der Tabelle
  - Anzahl Hashtabellen-Operationen (ohne Rehash)

$$\geq \frac{\alpha_0}{2} \cdot m$$

Kosten Rehash

- Die Gesamt-Rehash-Kosten sind dann

$$\leq \sum_{i=0}^{k-1} c \cdot m_0 \cdot 2^i = c \cdot m_0 \cdot \left(2^k - 1\right) \leq c \cdot m$$

- Anzahl Hashtabellen-Operationen

$$\#\mathrm{OP} \geq \frac{\alpha_0}{2} \cdot m$$

- Durchschnittskosten pro Operation

$$\frac{\#\mathbf{OP} \cdot \boldsymbol{c} + \mathbf{Rehash\_Kosten}}{\#\mathbf{OP}} \leq \boldsymbol{c} + \frac{\mathbf{2}\boldsymbol{c}}{\boldsymbol{\alpha_0}} \in \boldsymbol{O}(\mathbf{1})$$

- Im Durchschnitt sind die Kosten pro Operation konstant
  - auch für worst-case Eingaben (solange die Annahmen zutreffen)
  - **Durchschnittskosten pro Operation = amortisierte Kosten der Operation**

Amortisierte Analyse

Algorithmenanalyse bisher:

- worst case, best case, average case

Jetzt zusätzlich **amortized worst case**:

- n Operationen $o_1, \dots, o_n$ auf einer Datenstruktur, t_i: Kosten von o_i
- Kosten können sehr unterschiedlich sein (z.B. $t_i \in [1, c \cdot i]$)
- Amortisierte Kosten pro Operation

$$\frac{T}{n}, \qquad \text{wobei } T = \sum_{i=1}^{n} t_i$$

- **Amortisierte Kosten:** Durchschnittskosten pro Operation bei einer worst-case Ausführung
  - amortized worst case $\neq$ average case!

- Mehr dazu in der Algorithmentheorie-Vorlesung

Amortisierte Analyse Rehash

- Falls man immer nur vergrössert und davon ausgeht, dass bei kleinem Load, Hashtabellenop. $O(1)$ Kosten haben, sind die amortisierten Kosten pro Operation $O(1)$.
- Analyse funktioniert auch bei zufälliger Hashfunktion aus universeller Familie (mit hoher Wahrscheinlichkeit)
  - dann haben Hashtabellen-Op. bei kleinem Load mit hoher Wahrscheinlichkeit amortisierte Kosten $O(1)$
- Die Analyse lässt sich auch auf Rehashs zum Verkleinern erweitern
  - Und auch auf Fälle, wo man aufgrund von vielen delete-Operationen, einen Rehash machen muss
- In einer ähnlichen Art kann man aus fixed-size Arrays dynamische Arrays bauen
  - Alle Arrayoperationen haben dann $O(1)$ amortisierte Laufzeit
  - Vergrössern/verkleinern erlaubt der ADT nur in 1-Elem.-Schritten am Ende!

Cuckoo Hashing Idee

Hashing Zusammenfassung:

- effiziente Dictionary-Datenstruktur
- Operationen brauchen im Erwartungswert (meistens) $O(1)$ Zeit
- Hashing mit Chaining kann man so implementieren, dass insert immer $O(1)$ Laufzeit hat
- Können wir auch bei **find $\boldsymbol{O(1)}$ Laufzeit** garantieren?
  - wenn gleichzeitig insert nur noch im Erwartungswert $O(1)$ ist…

Cuckoo Hashing Idee:

- Offene Adressierung
  - an jeder Position der Tabelle hat es nur für ein Element Platz
- Zwei Hashfunktionen h_1 und h_2
- Ein Schlüssel x wird immer bei $h_1(x)$ oder $h_2(x)$ gespeichert
  - Falls beim Einfügen beide Stellen schon besetzt sind, müssen wir umorganisieren…

Cuckoo Hashing

Enfügen eines Schlüssels x:

- x wird immer an der Stelle $h_1(x)$ eingefügt
- Falls schon ein anderer Schlüssel y an der Stelle $h_1(x)$ ist:
  - Werfe y da raus (daher der Name: Cuckoo Hashing)
  - y muss an seiner alternativen Stelle eingefügt werden (falls es bei $h_1(y)$ war, an Stelle $h_2(y)$, sonst an Stelle $h_1(y)$)
  - falls da auch schon ein Element z ist, werfe z raus und platziere es an seiner Alternativposition
  - und so weiter…

Find / Delete:

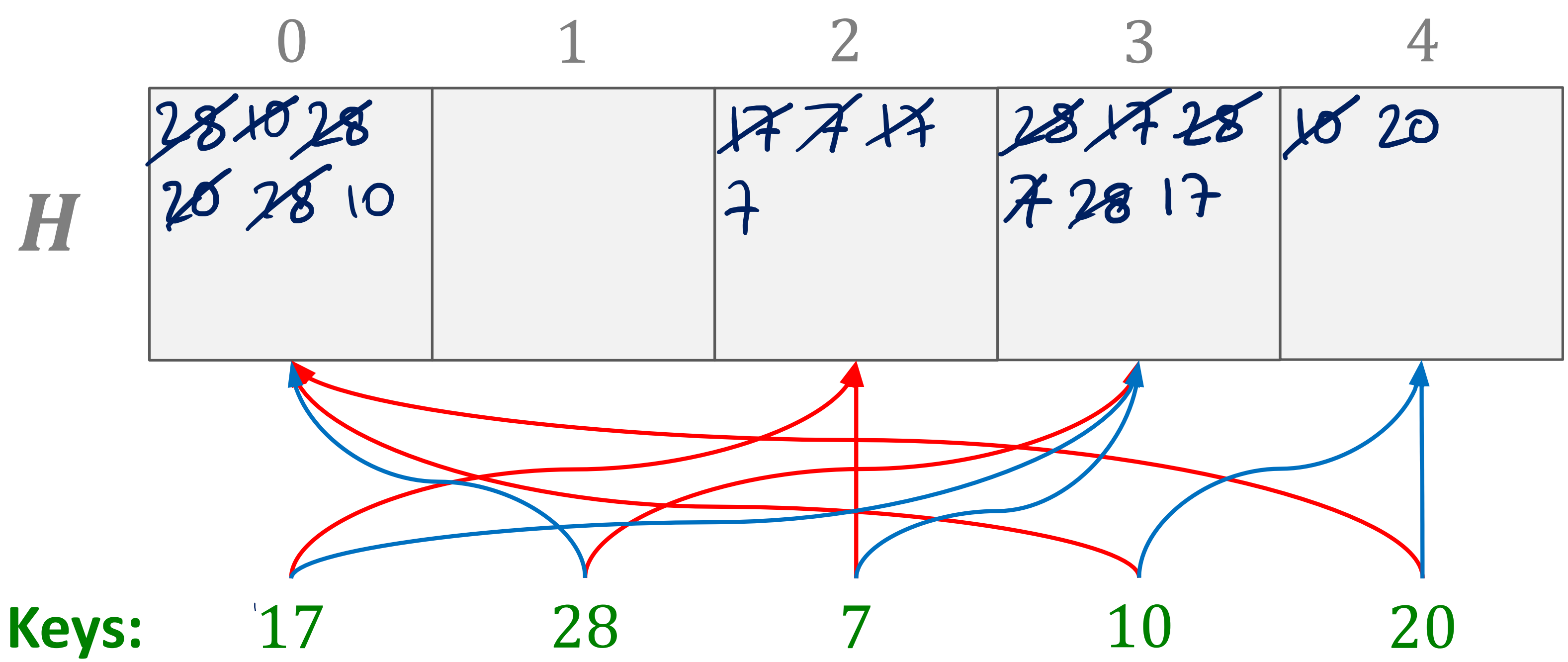
- Falls x in der Tabelle ist, ist's an Stelle $h_1(x)$ oder $h_2(x)$
- bei Delete: Markiere Zelle als leer!
- beide Operationen immer $O(1)$ Zeit!

Cuckoo Hashing Beispiel

Tabellengrösse $m = 5$

Hashfunktionen $h_1(x) = x \bmod 5$, $h_2(x) = 2x - 1 \bmod 5$

Füge Schlüssel 17, 28, 7, 10, 20 ein:



Cuckoo Hashing : Zyklen

- Beim Einfügen kann es zu einem Zyklus kommen
  - x wirft y_1 raus
  - y_1 wirft y_2 raus
  - y_2 wirft y_3 raus
  - …
  - $y_{\ell-1}$ wirft y_ℓ raus
  - y_ℓ wirft x oder y_i für $i < \ell$ raus
- Oder es kann irgendwann sein, dass $h_1(y_i) = h_2(y_i)$
- Dann wird noch der alternative Platz für x ausprobiert, aber da kann das Gleiche auch wieder passieren…
- In dem Fall wählt man neue Hash-Funktionen und macht einen Rehash (normalerweise mit grösserer Tabelle)

Cuckoo Hashing : Hashfunktionen

Wie wählt man die zwei Hashfunktionen?

- Sie sollten möglichst "unabhängig" sein...
- Wenige Schlüssel x, für welche $h_1(x) = h_2(x)$
- Eine gute Möglichkeit:

  Zwei unabhängige, zufällige Funktionen einer universellen Menge

- Dann kann man zeigen, dass Zyklen nur sehr selten vorkommen, solange $n \leq m/2$

- Sobald die Tabelle halbvoll ist ($n \geq m/2$) sollte man daher einen Rehash machen und zu einer doppelt so grossen Tabelle wechseln

Cuckoo Hashing : Laufzeit

Find / Delete:

- Hat immer Laufzeit $O(1)$
- Man muss nur die zwei Stellen $h_1(x)$ und $h_2(x)$ anschauen
- Das ist der grosse Vorteil von Cuckoo Hashing

Insert:

- Man kann zeigen, dass das im Durchschnitt auch Zeit $O(1)$ braucht
- Falls man die Tabelle nicht mehr als zur Hälfte füllt
- Verdoppeln der Tabellengrösse bei Rehash ergibt konstante durchschnittliche Laufzeit für alle Operationen!

Zusammenfassung Hashing

effiziente Methode, um einen Dictionary zu implementieren

Behandeln von Kollisionen

- Hashing mit Chaining
  - Einfach, sehr flexibel, mit 2 Hash Funktionen kann man die Listenlängen mit hoher Wahrscheinlichkeit auf $O(\log\log n)$ beschränken
- Offene Adressierung
  - Verschiedene Möglichkeiten, in der Praxis typischerweise effizienter
  - Es ist möglich, find in worst-case $O(1)$ Zeit zu implementieren
  - Load $\alpha > 1$ nicht möglich, falls α gross wird, muss man einen Rehash machen

Hashfunktionen

- Es gibt Strategien, um einfach gute Hashfunktionen zu erhalten
  - In der Praxis wird oft eine fixe, recht einfache Funktion verwendet

Rehash

- Wenn die Hashtabelle zu voll wird, muss man neu aufsetzen
  - Das kann man so machen, dass die Gesamtlaufzeit pro Hashtabellenoperation konstant bleibt