Informatik II - SS 2018

(Algorithmen & Datenstrukturen)

Vorlesung 7 (9.5.2018)

Hashtabellen II

UNI FREIBURG

Fabian Kuhn

Algorithmen und Komplexität

Bemerkungen Übungsblatt 2

UNI FREIBURG

- Viele haben Mühe, formal sauber zu argumentieren
  - war bei Aufgabe 1 notwendig

- Aufgabe 3: Quicksort Pivotwahl
  - Oft wurde einfach angegeben, was die Eigenschaften der Eingabe sind, damit die Pivotwahl möglichst gut/schlecht ist, statt wirklich eine Eingabe explizit anzugeben.

- Offenbar haben einige nicht gewusst, was eine Rekursionsgleichung ist
  - Da hatten wir in der Vorlesung mehrere Beispiele
  - Wenn's trotzdem nicht klar ist → Forum

Bemerkungen zum Übungsblatt 2

UNI FREIBURG

Aufgabe 1 (Beweise mit Hilfe der Def. O-Notation)

1b) $n \in \Omega(\log_2 3^n)$

$f(n) \in O(g(n)) : \exists c, n_0 \; \forall n \geq n_0 \; f(n) \leq c\, g(n)$

$f(n) \in \Omega(g(n)) : \exists c, n_0 \; \forall n \geq n_0 \; f(n) \geq c\, g(n)$

$\log_2 3^n = n \cdot \underbrace{\log_2 3}_{a}$

$n \geq c \cdot a \cdot n$

$c = (\log_2 3)^{-1}, \; n_0 = 1$

$n \geq c \cdot \log_2 3^n$

1c) $2n \in O(10\sqrt{n})$

$\forall c, n_0 \; \exists n \geq n_0 \; f(n) > c \cdot g(n)$

$n \leq 5 \cdot c \cdot \sqrt{n}$

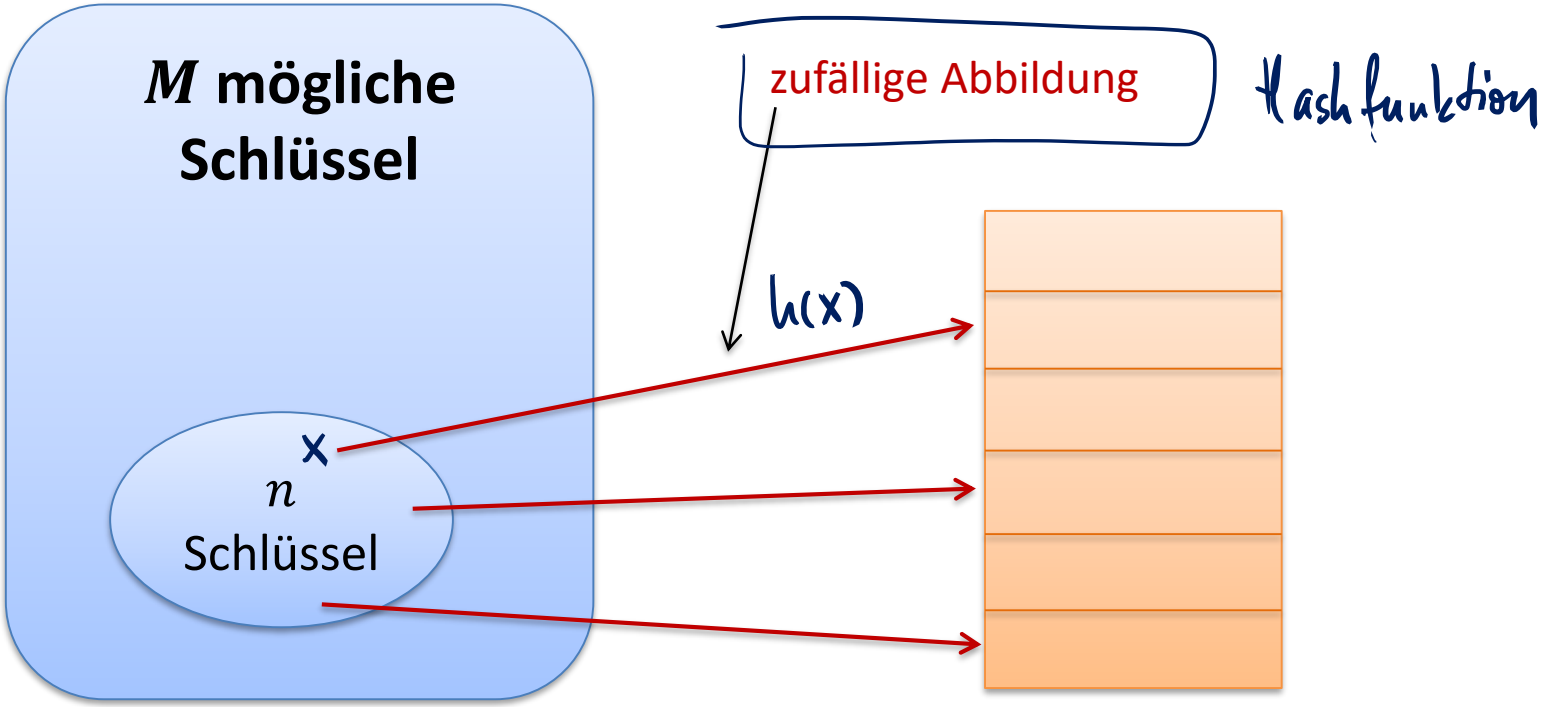
(*) $5 \cdot c \geq \sqrt{n}$ gilt nicht für genug grosse n

egal, wie n_0 gewählt ist, gibt es ein $n \geq n_0$, so dass (*) nicht gilt.

Hashing : Idee

Problem

- Riesiger Raum S an möglichen Schlüsseln
- Anzahl n der wirklich benutzten Schlüssel ist **viel** kleiner
  - Wir möchten nur Arrays der Grösse $\approx n$ (resp. $O(n)$) verwenden…
- Wie können wir M Schlüssel auf $O(n)$ Array-Positionen abbilden?



Hashfunktionen

UNI FREIBURG

Schlüsselraum S, $|S| = M$ (alle möglichen Schlüssel)

Arraygrösse m ($\approx$ Anz. Schlüssel, welche wir max. speichern wollen)

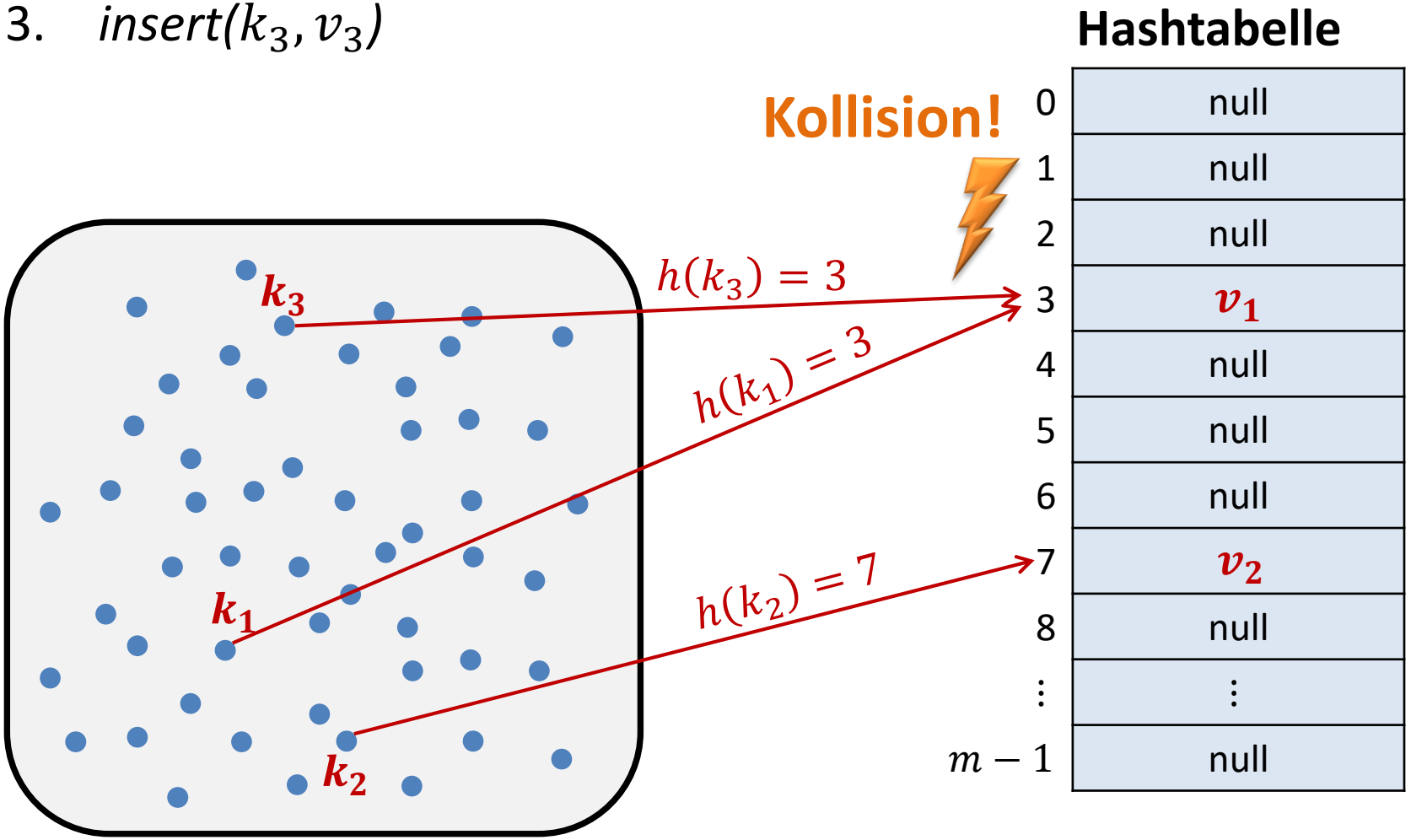
Hashfunktion

$$h: S \to \{0, \dots, m-1\}$$

- Bildet Schlüssel vom Schlüsselraum S in Arraypositionen ab
- h sollte möglichst nahe bei einer zufälligen Funktion sein
  - alle Elemente in $\{0, \dots, m-1\}$ etwa gleich vielen Schlüsseln zugewiesen sein
  - ähnliche Schlüssel sollten auf verschiedene Positionen abgebildet
- h sollte möglichst schnell berechnet werden können
  - Wenn möglich in Zeit $O(1)$
  - Wir betrachten es im folgenden als Grundoperation (Kosten = 1)

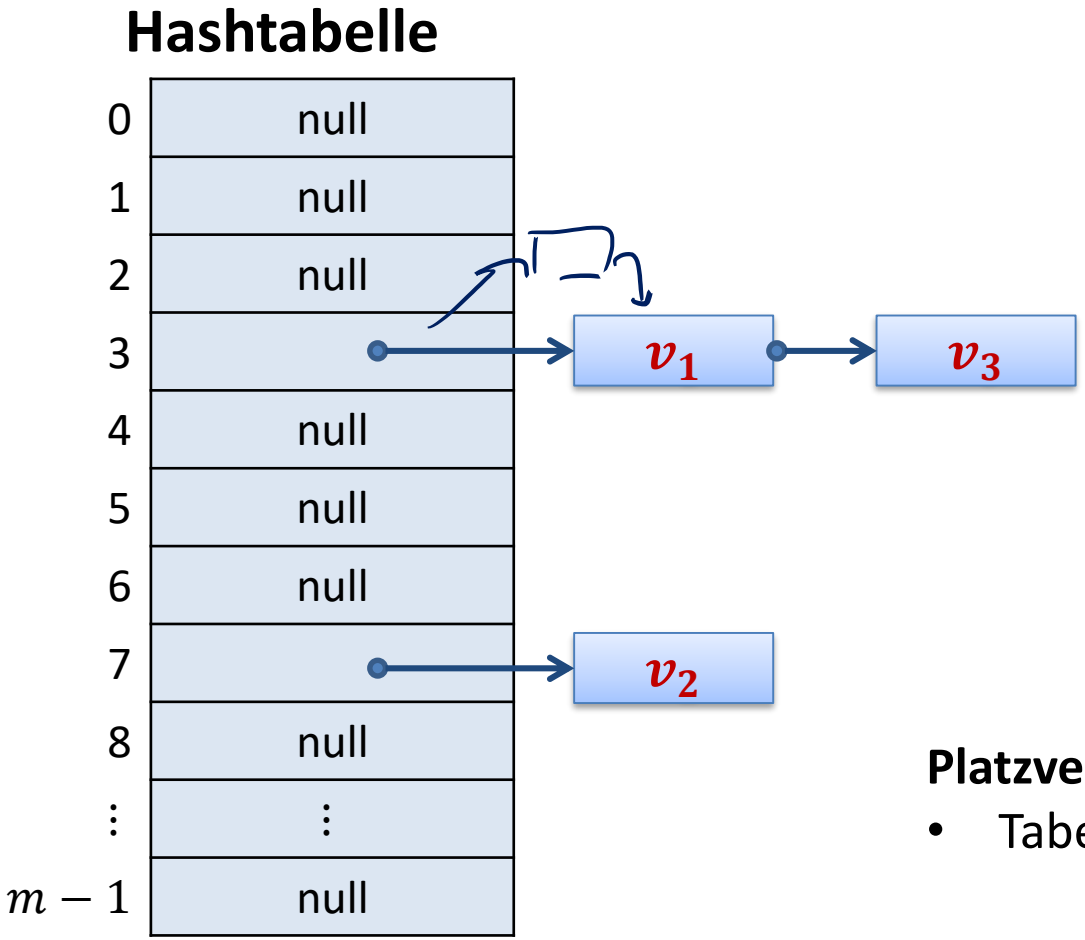
Funktionsweise Hashtabellen

1. *insert*(k_1, v_1)
2. *insert*(k_2, v_2)
3. *insert*(k_3, v_3)



Hashtabellen mit Chaining

- Jede Stelle in der Hashtabelle zeigt auf eine verkette Liste



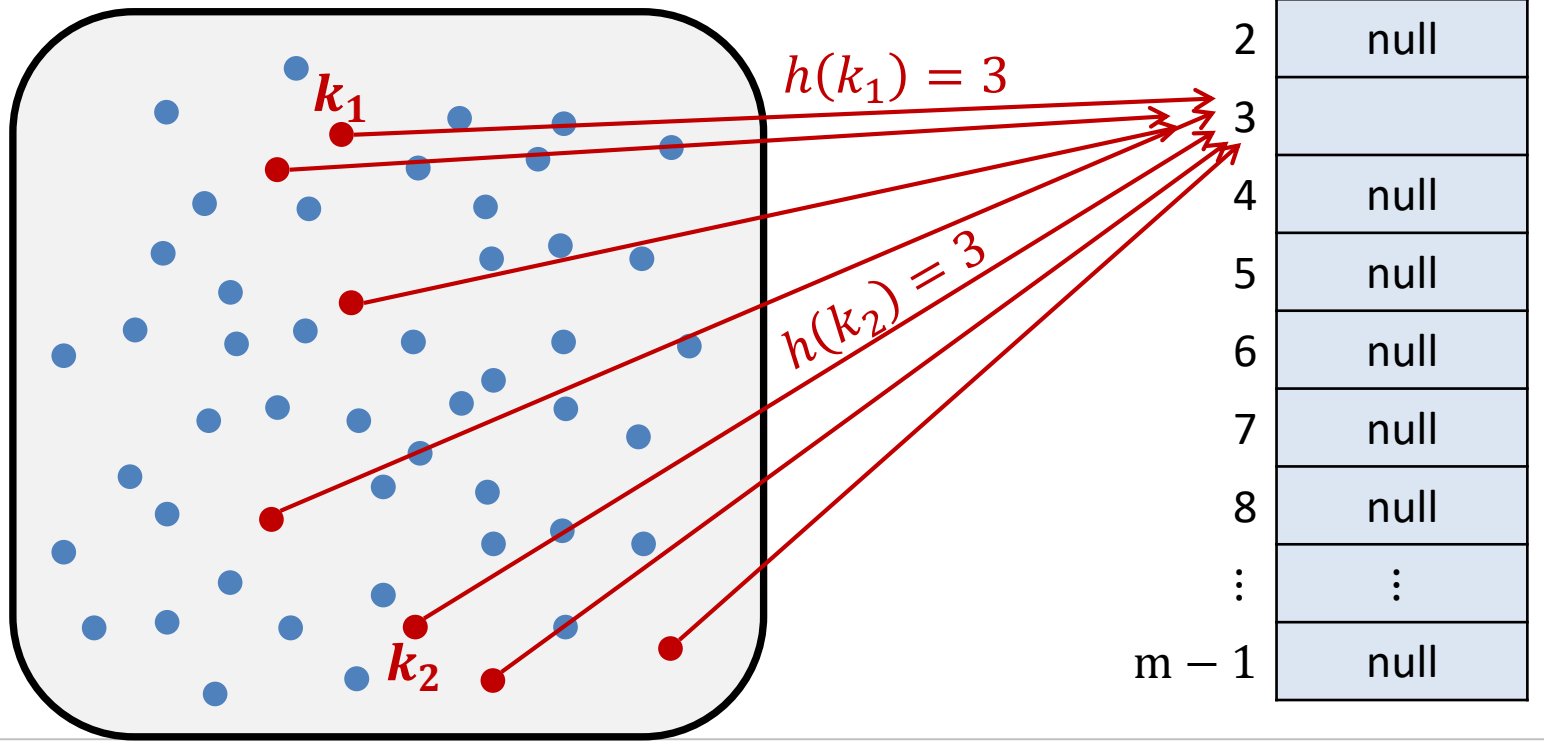
Platzverbrauch:

- Tabellengrösse m, Anz. Elemente n

Funktionsweise Hashtabellen

Schlechtester Fall bei Hashing mit Chaining

- Alle Schlüssel, welche vorkommen, haben den gleichen Hashwert
- Ergibt eine verkettete Liste der Länge n
- Wahrscheinlichkeit bei zufälligem h: $\frac{1}{m^{n-1}}$



Länge der verketten Liste

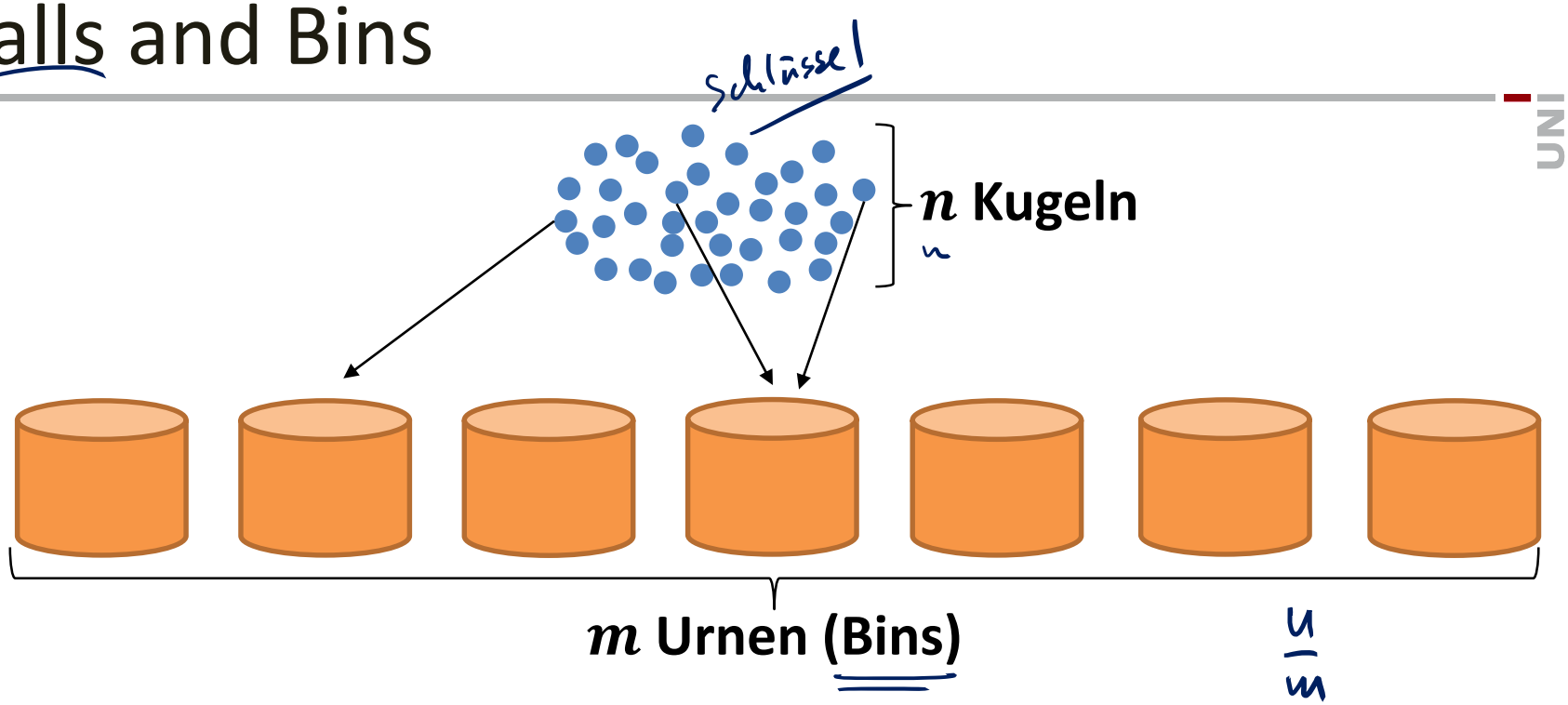
- Kosten von *find* und *delete* hängt von der Länge der entprechenden Liste ab
- Wie lang werden die Listen
  - Annahme: Grösse der Hashtabelle m, Anzahl Elemente n
  - Weitere Annahme: Hashfunktion h verhält sich wie zufällige Funktion
- Listenlängen entspricht folgendem Zufallsexperiment

m Urnen und n Kugeln

- Jede Kugel wird (unabhängig) in eine zufällige Urne geworfen
- Längste Liste = maximale Anz. Kugeln in der gleichen Urne
- Durchschnittliche Listenlänge = durchschn. Anz. Kugeln pro Urne

m Urnen, n Kugeln → durschn. #Kugeln pro Urne: n/m

Balls and Bins



- Worst-case Laufzeit = $\Theta(\max \#\text{Kugeln pro Urne})$

  mit hoher Wahrscheinlichkeit $\in O\left(n/m + \log n/\log\log n\right)$

  - bei $n \leq m$ also $O\left(\log n/\log\log n\right)$

- Erwartete Laufzeit (für jeden Schlüssel):
  - entpricht der #Kugeln in der Urne einer zufälligen Kugel

Erwartete Laufzeit von Find

UNI FREIBURG

Load α der Hashtabelle:

$$\alpha := \frac{n}{m}$$

Kosten einer Suche:

- Suche nach einem Schlüssel x, welcher nicht in der Hashtabelle ist

  $h(x)$ ist eine uniform zufällige Position

  → erwartete Listenlänge = durchschn. Listenlänge = α

Erwartete Laufzeit: $1 + \alpha$

Laufzeiten Hashing mit Chaining

UNI FREIBURG

Zusammenfassung Laufzeiten:

create & insert:

- Immer Laufzeit $O(1)$ (auch im Worst Case, unabhängig von α)

find & delete:

$\alpha = O(1)$

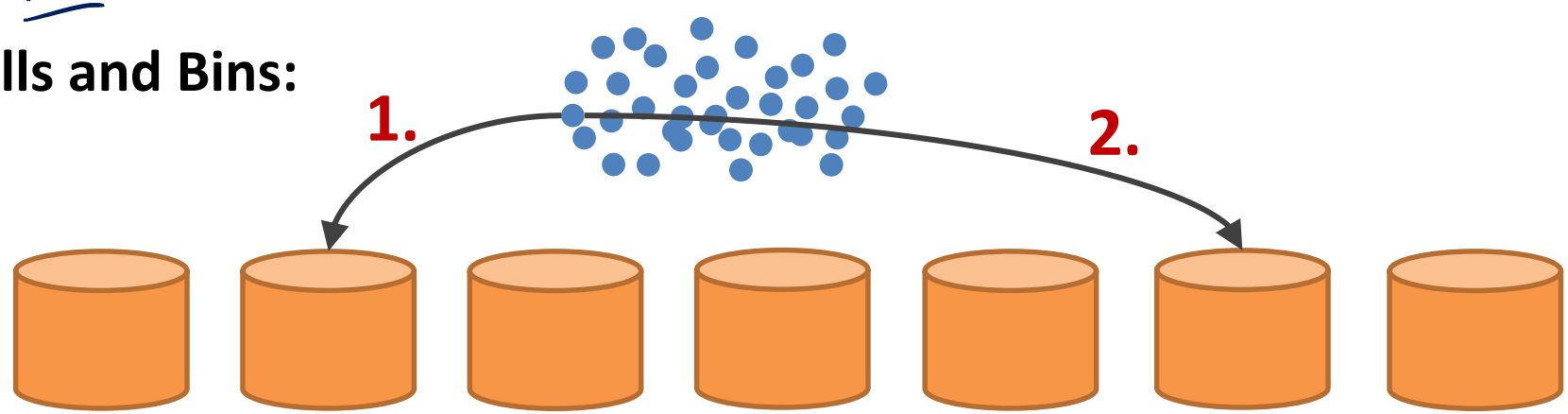
- Worst Case: $\boldsymbol{\Theta(n)}$
- Worst Case mit hoher Wahrsch. (bei zufälligem h): $\boldsymbol{O\left(\alpha + \frac{\log n}{\log \log n}\right)}$
- Erwartete Laufzeit (für bestimmten Schlüssel x): $\boldsymbol{O(1+\alpha)}$
  - gilt für erfolgreiche und nicht erfolgreiche Suchen
  - Falls $\alpha = O(1)$ (d.h., Hashtabelle hat Grösse $\Omega(n)$), dann ist das $O(1)$
- Hashtabellen sind extrem effizient und haben **typischerweise $\boldsymbol{O(1)}$ Laufzeit für alle Operationen**.

Kürzere Listenlängen

Idee:

- Benutze zwei Hashfunktionen h_1 und h_2
- Füge Schlüssel x in die kürzere der beiden Listen bei $h_1(x)$ und $h_2(x)$ ein

Balls and Bins:



- Lege Kugel in Urne mit weniger Kugeln
- Bei n Kugeln, m Urnen: maximale Anz. Kugeln pro Urne (whp):

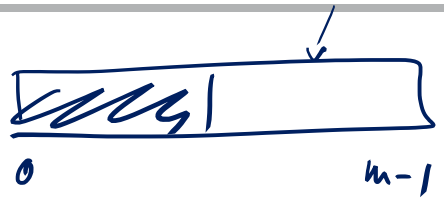
$$\boldsymbol{n/m + O(\log\log m)}$$

- Bekannt als "power of two choices"

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 von verschiedenen Schlüsseln ist unabhängig
    (nicht klar, was das bei einer deterministischen Funktion genau heissen soll)
  - Mapping ist uniform zufällig (alle Hashwerte kommen gleich oft vor)
- 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

$X = i, i+m, i+2m, \ldots$

1000 Elemente

Wähle Hashfunktion als

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

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

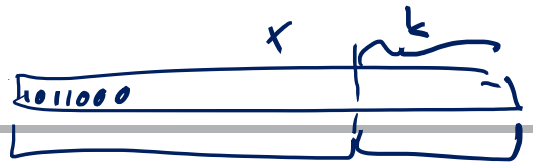
Vorteile:

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

Bemerkung:

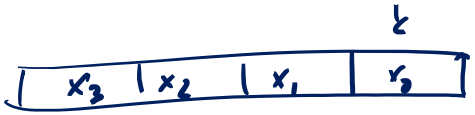
- Falls die Schlüssel keine ganzen Zahlen sind, kann man den Bitstring als ganze Zahl interpretieren

Divisionsmethode



Wähle Hashfunktion als

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



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

$$x \cdot (2^k) \bmod (2^k - 1) = x \bmod (2^k - 1)$$

$$X = x_0 + x_1 \cdot (\dots) + x_2 \cdot (\dots)^2 + \dots$$

- 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

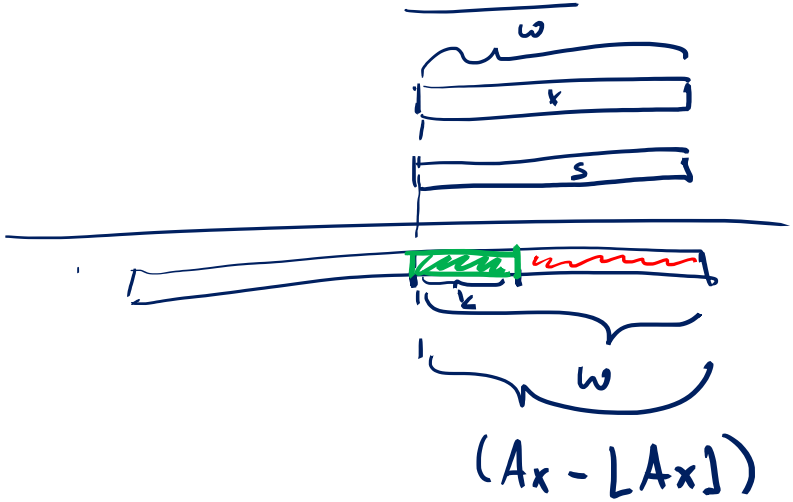
$$h(x) = \lfloor m \cdot (Ax - \lfloor Ax \rfloor) \rfloor$$

- 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 $s \in \{0, \dots, 2^w - 1\}$ 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 $s \in \{0, \dots, 2^w - 1\}$ 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:

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

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 I

Definition:

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

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

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

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

Universelles Hashing II

Theorem: $|X| = n$

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

- Für ein gegebenes $x \in X$ sei

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

$\frac{n}{m} = \alpha$

- Im Erwartungswert hat B_x Grösse $\leq 1 + c \cdot \frac{|X|}{m} = 1 + c\frac{n}{m} = 1 + c \cdot \alpha$

Konsequenz:

- Im Erwartungswert sind alle Listen kurz!

Universelles Hashing III

- Gute universelle Mengen von Hashfunktionen existieren!

Beispiele:

- m beliebig, p: Primzahl mit $p > m$ und $p \geq |S|$

  $\mathcal{H}$: Menge der Fkt. $h_{a,b}(x) = (a \cdot x + b) \bmod p \bmod m$

  – wobei $a, b \in S$

  Parameter

  c-universell für $c \approx 1$

  $a = a_0 + a_1 \cdot 10 + a_2 \cdot 10^2 + \ldots$

  $a = a_0 + a_1 \cdot m + a_2 m^2 + \ldots$

  $a_i \in \{0, \ldots, m-1\}$

- m beliebig, $k = \lceil \log_m |S| \rceil$, Parameter $a \in S$

  Basis m-Darstellung von a, x: $a = \sum_{i=0}^{k-1} a_i \cdot m^i$, $x = \sum_{i=0}^{k-1} x_i \cdot m^i$

  $\mathcal{H}$: Menge der Fkt. $h_a(x) = \left(\sum_{i=0}^{k-1} a_i \cdot x_i\right) \bmod m$

  1-universell

Hashing mit offener Adressierung

Ziel:

- Speichere alles direkt in der Hashtabelle (im Array)
- offene Adressierung = geschlossenes Hashing
- keine Listen

Grundidee:

- Bei Kollisionen müssen alternative Einträge zur Verfügung stehen
- Erweitere Hashfunktion zu $h(x)$ / $h(x,0), h(x,1), h(x,2), \ldots$

$$h: S \times \{0, \ldots, m-1\} \to \{0, \ldots, m-1\}$$

  - Für jedes $x \in S$ sollte $h(x, i)$ durch alle m Werte gehen (für versch. i)

- Zugriff (schreiben/lesen) zu Element mit Schlüssel x:
  - Versuche der Reihe nach an den Positionen

$$h(x,0), h(x,1), h(x,2), \ldots, h(x, m-1)$$

Lineares Sondieren

Idee:

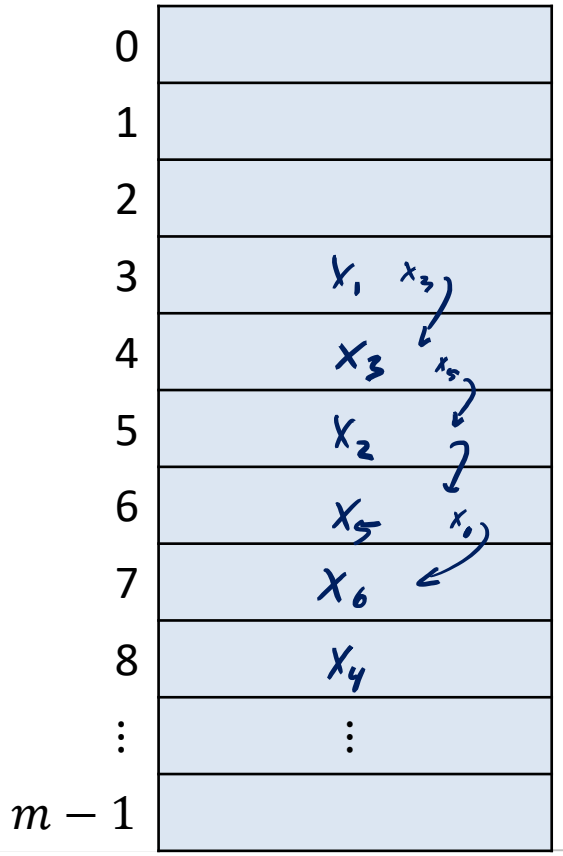
$h(x) = 17$

- Falls $h(x)$ besetzt, versuche die nachfolgende Position:

$$\boldsymbol{h(x,i) = (h(x) + i) \bmod m}$$

für $i = 0, \ldots, m-1$

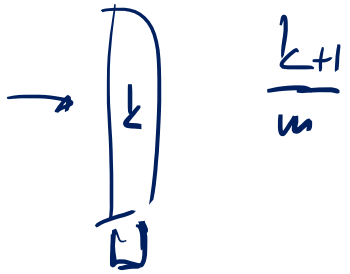
- **Beispiel:**
  Füge folgende Schlüssel ein
  - $x_1, h(x_1) = 3$
  - $x_2, h(x_2) = 5$
  - $x_3, h(x_3) = 3$
  - $x_4, h(x_4) = 8$
  - $x_5, h(x_5) = 4$
  - $x_6, h(x_6) = 6$
  - …



Lineares Sondieren

Vorteile:

- sehr einfach zu implementieren
- alle Arraypositionen werden angeschaut
- gute Cache-Lokalität



Nachteile:

- Sobald es Kollisionen gibt, bilden sich Cluster
- Cluster wachsen, wenn man in irgendeine Position des Clusters "hineinhasht"
- Cluster der Grösse k wachsen in jedem Schritt mit Wahrscheinlichkeit $(k+1)/m$
- Je grösser die Cluster, desto schneller wachsen sie!!

Quadratisches Sondieren

Idee:

- Nehme Sequenz, welche nicht zu Cluster führt:

$$h(x,i) = \left(h(x) + c_1 i + c_2 i^2\right) \bmod m$$

für $i = 0, \dots, m-1$

$h(x) = h(y) \Rightarrow h(x,i) = h(y,i)$

Vorteil:

- ergibt keine zusammenhängenden Cluster
- deckt bei geschickter Wahl der Parameter auch alle m Positionen ab

Nachteil:

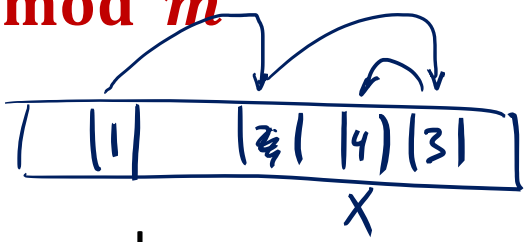
- kann immer noch zu einer Art Cluster-Bildung führen
- Problem: der erste Hashwert bestimmt die ganze Sequenz!
- Asympt. im besten Fall so gut, wie Hashing mit verketteten Listen

Doppel-Hashing

Ziel: Verwende mehr als m verschiedene Abfolgen von Positionen

Idee: Benutze zwei Hashfunktionen

$$h(x,i) = \big(h_1(x) + i \cdot h_2(x)\big) \bmod m$$



Vorteile:

- Sondierungsfunktion hängt in zwei Arten von x ab
- Vermeidet die Nachteile von linearem und quadr. Sondieren
- Wahrscheinlichkeit, dass zwei Schlüssel x und x' die gleiche Positionsfolge erzeugen:

$$h_1(x) = h_1(x') \wedge h_2(x) = h_2(x') \implies \text{WSK} = \frac{1}{m^2}$$

- Funktioniert in der Praxis sehr gut!

Offene Adressierung : Zusammenfassung

UNI FREIBURG

Offene Adressierung:

- Alle Schlüssel/Werte werden direkt im Array gespeichert
- Keine Listen nötig
  - spart den dazugehörigen Overhead…
- Nur schnell, solange der Load

$$\alpha = \frac{n}{m} \qquad \alpha \leq \frac{1}{2}$$

  nicht zu gross wird…
  - dann ist's dafür besser als Chaining…
- $\alpha > 1$ ist nicht möglich!
  - da nur m Positionen zur Verfügung stehen

Rehash

Was tun, wenn die Hashtabelle zu voll wird?

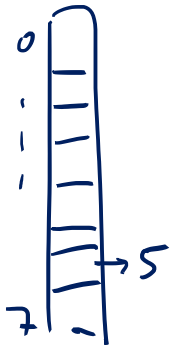
- Offene Adressierung: $\alpha > 1$ nicht möglich, bei $\alpha \to 1$ sehr ineff.
- Chaining: Komplexität wächst linear mit α

Was tun, wenn die gewählte Hashfunktion schlecht ist?

Rehash:

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

Beispiel: $X = \{5, 10, 11, 22\}$, $h(x) = x \bmod 4$, $h'(x) = 3x - 1 \bmod 8$



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.
- **richtig heisst:**
  - gute Hashfunktion (z.B. aus einer universellen Klasse)
  - gute Wahl der Tabellengrössen:

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

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

    → amortisierte Analyse