

# Informatik II - SS 2016

## (Algorithmen & Datenstrukturen)

Vorlesung 8 (13.5.2016)

### Hashtabellen I



**UNI  
FREIBURG**

Fabian Kuhn

Algorithmen und Komplexität

# Dictionary mit sortiertem Array

## Laufzeiten:

*create*:  $O(1)$

*insert*:  $O(n)$

*find*:  $O(\log n)$  binäre Suche

*delete*:  $O(n)$

Können wir alle Operationen schnell machen?

- und das *find* noch schneller?

# Direkte Adressierung

Mit einem Array können wir alles schnell machen,  
...falls das Array gross genug ist.

**Annahme:** Schlüssel sind ganze Zahlen zwischen 0 und  $M - 1$

0	None
1	None
→ 2	<u>Value 1</u>
3	None
→ 4	None
5	None
→ 6	Yannic
7	Value 3
8	None
⋮	⋮
<u><math>M - 1</math></u>	None

$find(\underline{2}) \rightarrow$  "Value 1"

$insert(\underline{6}, \text{"Yannic"})$

$delete(4)$

Kosten:

$O(1)$

## 1. Direkte Adressierung benötigt zu viel Platz!

- Falls Schlüssel ein beliebiger *int* (32 bit) sein kann:  
Wir benötigen ein Array der Grösse  $\underline{2^{32}} \approx \underline{4 \cdot 10^9}$ .  
Bei 64 bit Integers sind's sogar schon mehr als  $10^{19}$ ...

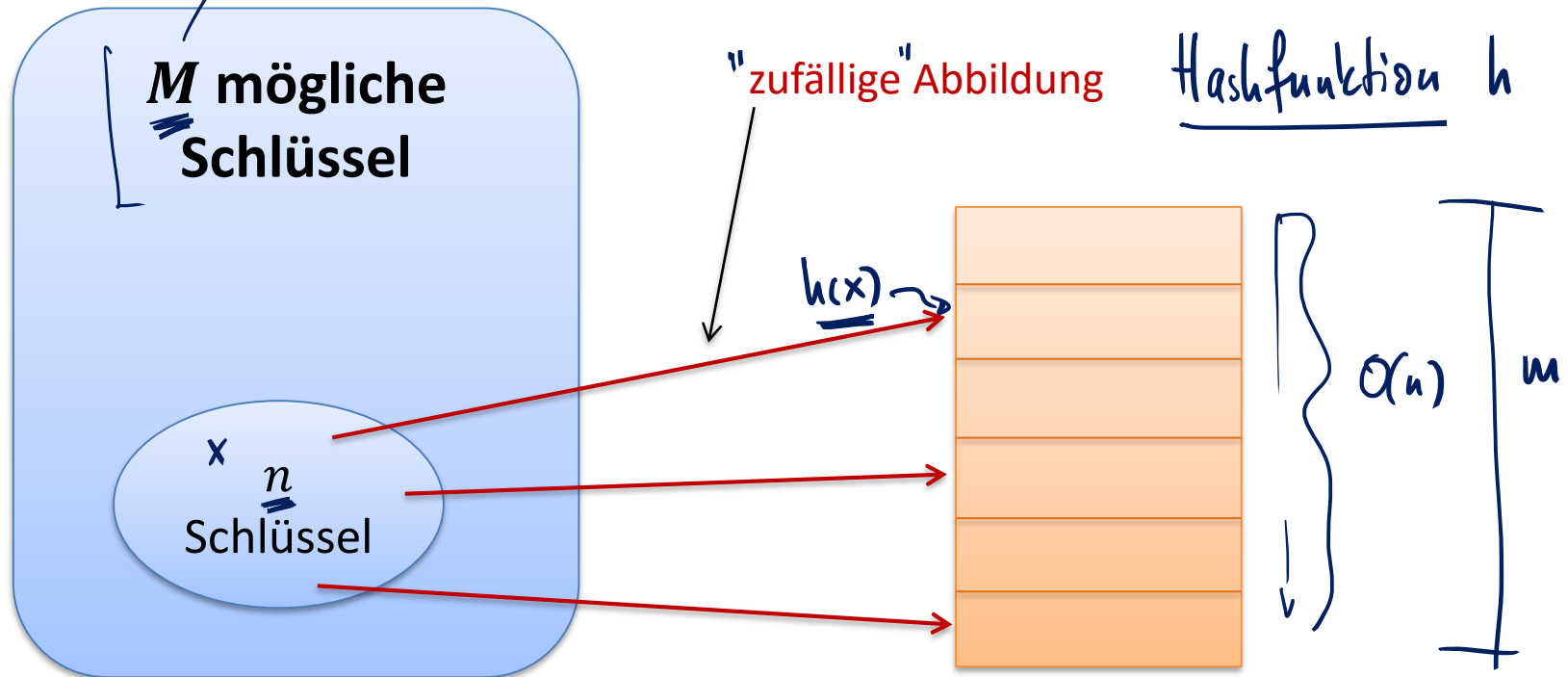
## 2. Was tun, wenn die Schlüssel keine ganzen Zahlen sind?

- Wo kommt das *(key,value)*-Paar ("*Alex*", "*Assistent*") hin?
- Wo soll der Schlüssel 3.14159 gespeichert werden?
- Pythagoras: "Alles ist Zahl"  
"Alles" kann als Folge von Bits abgespeichert werden:  
**Interpretiere Bit-Folge als ganze Zahl**
- **Verschärft das Platz-Problem noch zusätzlich!**

# 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?



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

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

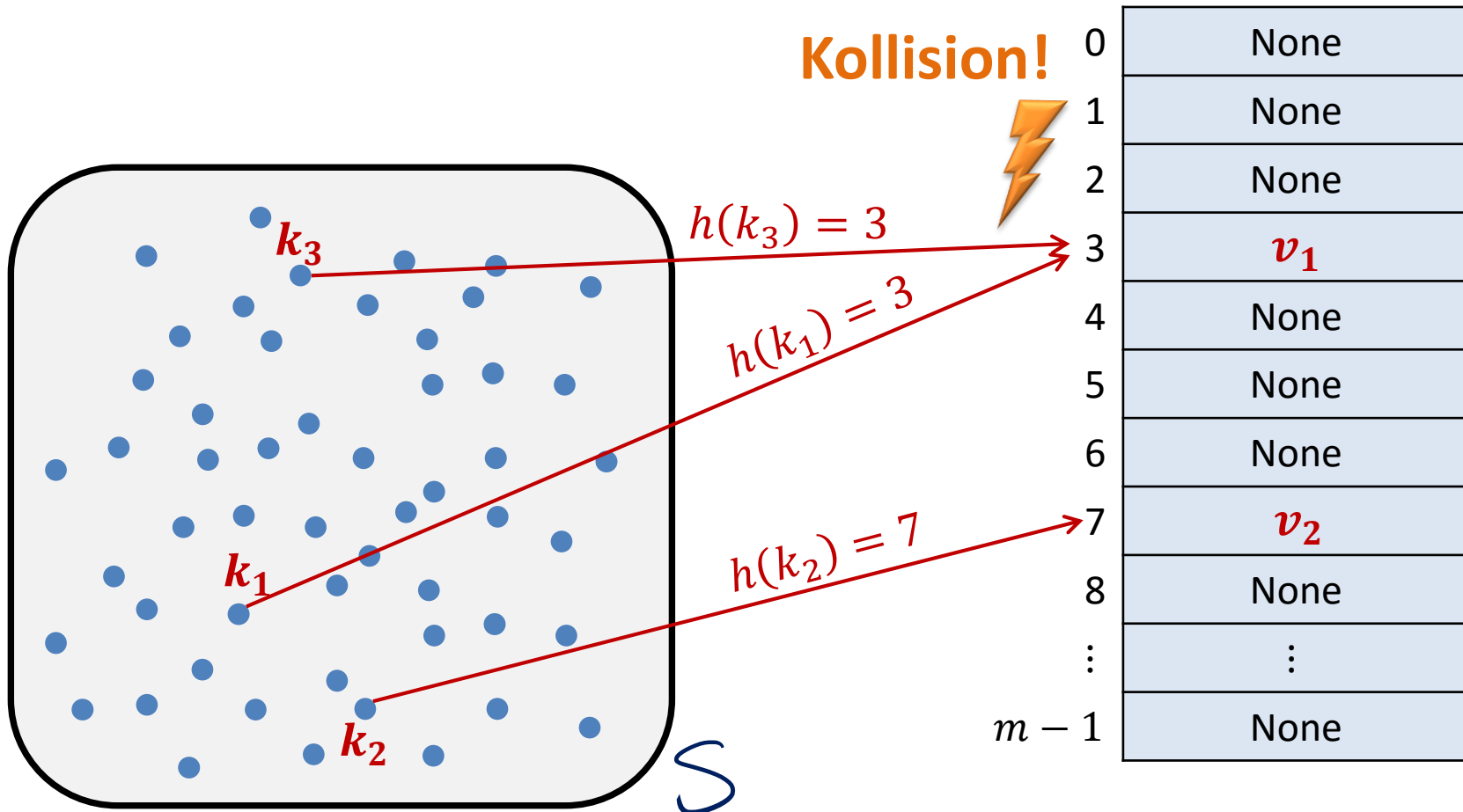
## Hashfunktion

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

- 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.  $insert(\underline{k}_1, \underline{v}_1)$
2.  $insert(k_2, v_2)$
3.  $insert(k_3, v_3)$



## Kollision:

Zwei Schlüssel  $k_1, k_2$  kollidieren, falls  $h(k_1) = h(k_2)$ .

## Was tun bei einer Kollision?

- Können wir Hashfunktionen wählen, bei welchen es keine Kollisionen gibt?

nein!  
(nur falls die Schlüssel bekannt sind)

- Eine andere Hashfunktion nehmen?

ja

- Weitere Ideen?

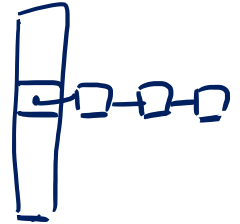


## Kollisionen Lösungsansätze

- Annahme: Schlüssel  $k_1$  und  $k_2$  kollidieren

### 1. Speichere beide (key,value)-Paare an die **gleiche Stelle**

- Die Hashtabelle muss an jeder Position Platz für mehrere Elemente bieten
- Wir wollen die Hashtabelle aber nicht einfach vergrößern (dann könnten wir gleich mit einer grösseren Tabelle starten...)
- **Lösung: Verwende verkettete Listen**

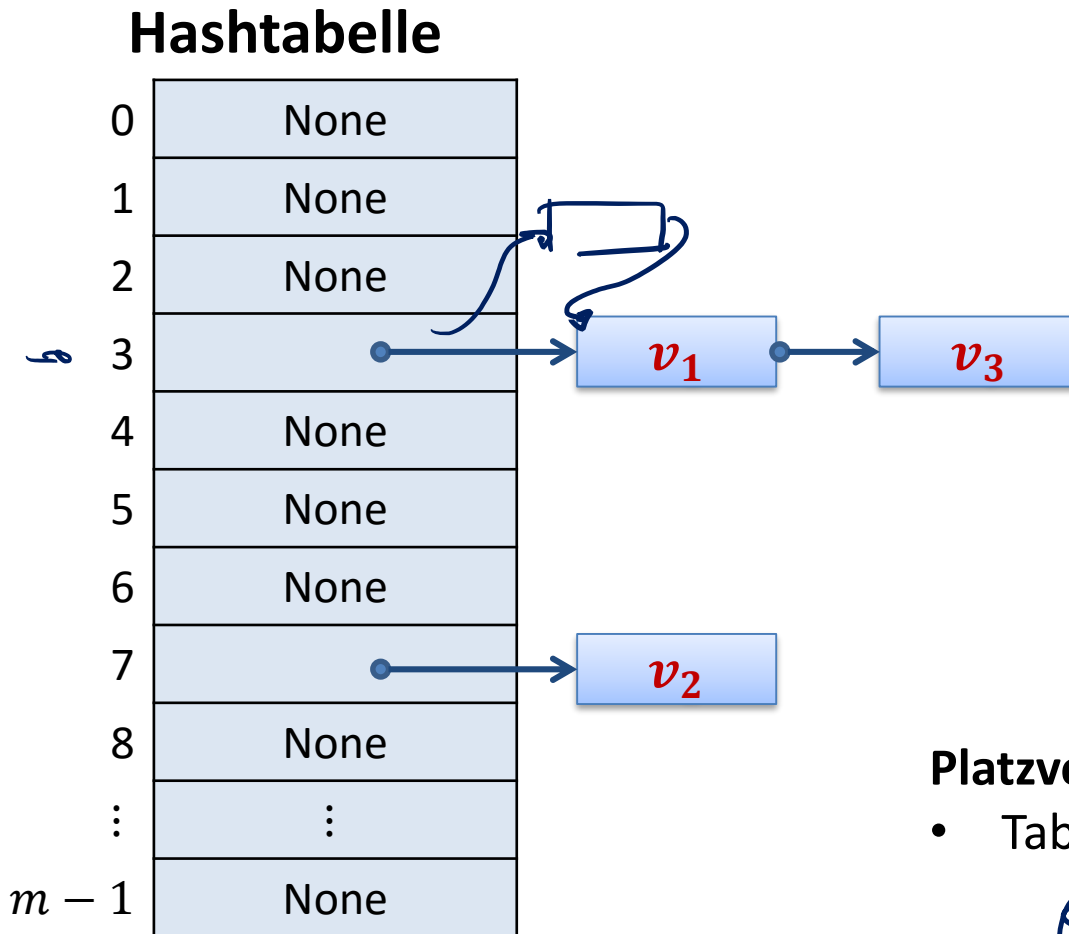


### 2. Speichere zweiten Schlüssel an eine **andere Stelle**

- Kann man zum Beispiel mit einer zweiten Hashfunktion erreichen
- Problem: An der alternativen Stelle könnte wieder eine Kollision auftreten
- Es gibt mehrere Lösungen
- Eine Lösung: Verwende **viele mögliche neue Stellen** (Man sollte sicherstellen, dass man die meistens nicht braucht...)

# Hashtabellen mit Chaining

- Jede Stelle in der Hashtabelle zeigt auf eine verkettete Liste



$insert(x, v)$

füge  $(x, v)$  in Liste  
an Pos.  $h(x)$  ein

$find(x)$

gehe zu Liste an  
Stelle  $h(x)$

$delete(x)$

**Platzverbrauch:**

- Tabellengröße  $m$ , Anz. Elemente  $n$

$$O(m + n)$$

Zuerst, um's einfach zu machen, für den Fall ohne Kollisionen...

*create:*  $O(1)$  /  $O(m)$

*insert:*  $O(1)$

*find:*  $O(1)$

*delete:*  $O(1)$

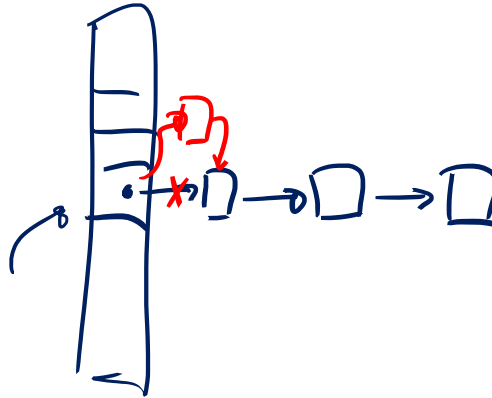
- Solange keine Kollisionen auftreten, sind Hashtabellen extrem schnell (falls die Hashfunktion schnell ausgewertet werden kann)
- Wir werden sehen, dass dies auch mit Kollisionen gilt...

# Laufzeit mit Chaining

Verkettete Listen an allen Positionen der Hashtabelle

create:  $O(1)$  ?

insert:  $O(1)$



$$\underline{\underline{M \geq n}}$$

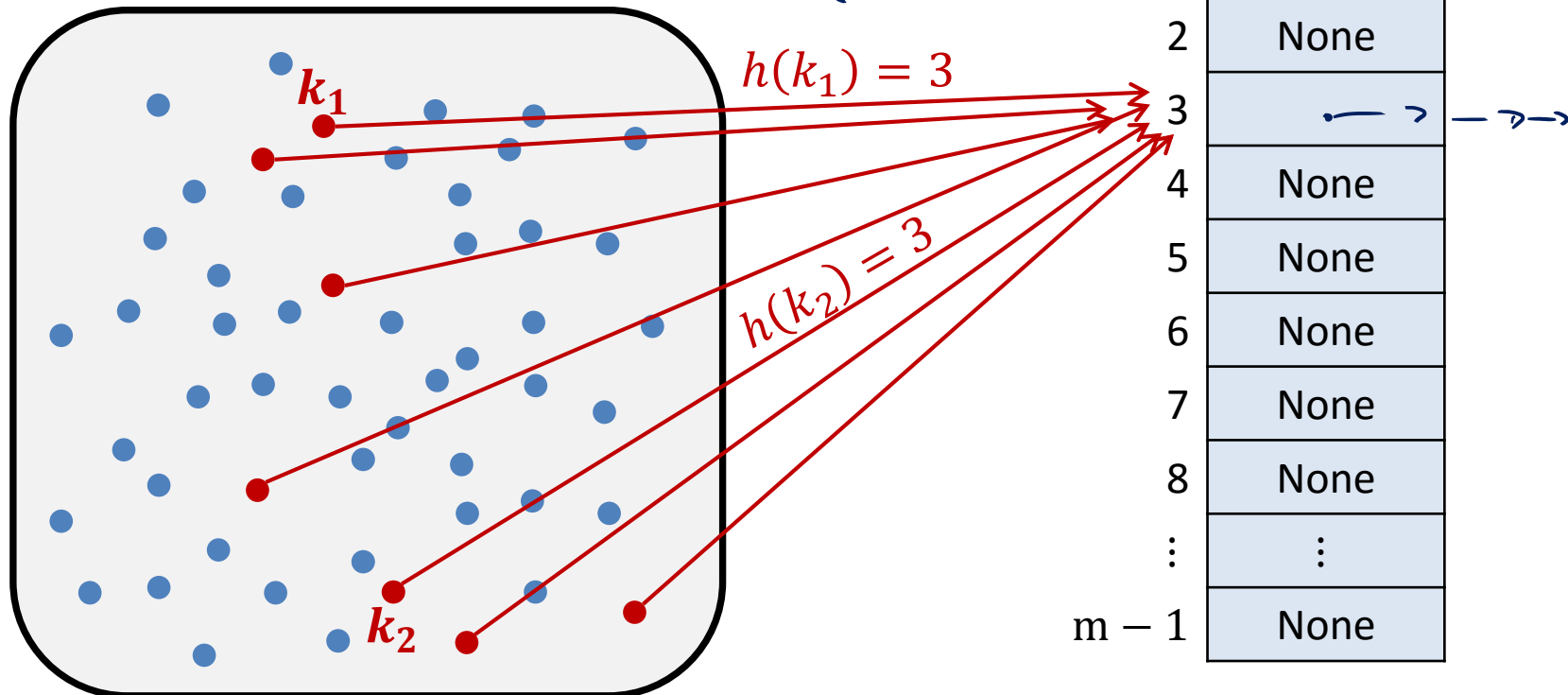
find:  $O(1 + \text{"Länge der Liste an Pos. } h(x)\text{"})$   
find(x)  
↑ im Durchschnitt  $\frac{n}{m}$

delete:  $\parallel$

# 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  $h$ :  $\left(\frac{1}{m}\right)^{n-1}$



# Länge der verketteten Liste

- Kosten von *find* und *delete* hängt von der Länge der entsprechenden 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

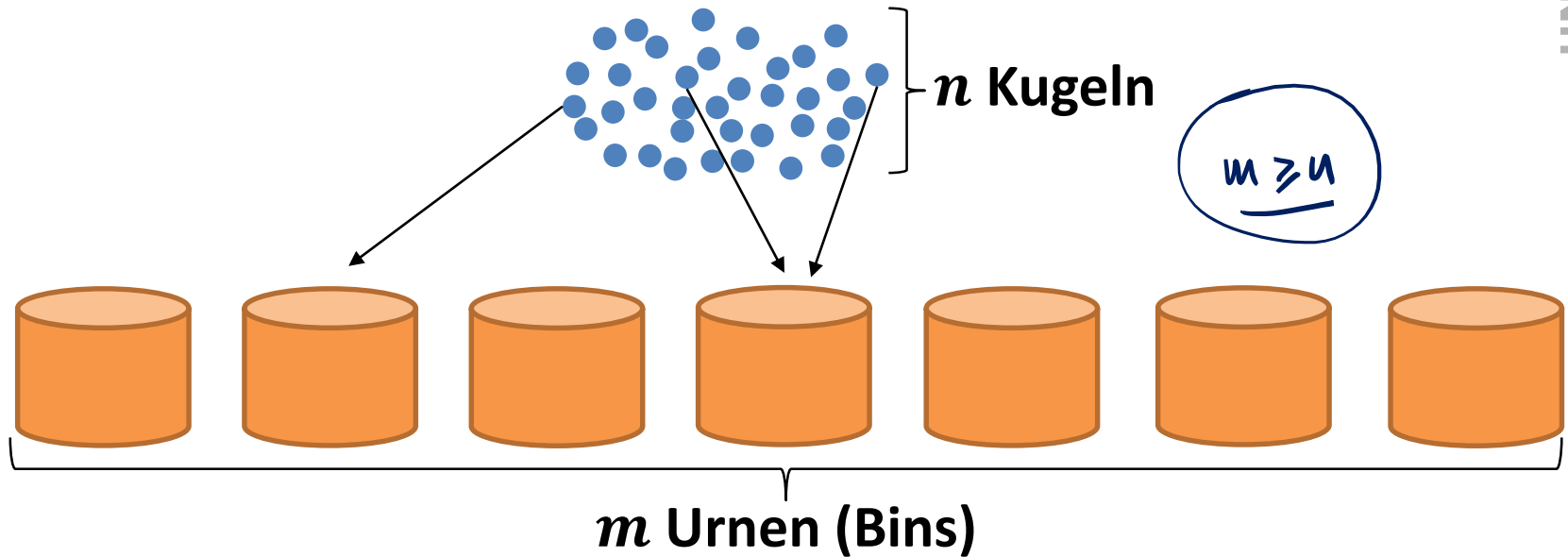
$m$  Urnen und  $n$  Kugeln

*bins* (under  $m$ )      *balls* (under  $n$ )

*Schlüssel* (arrow from  $n$  to  $m$ )      *Array pos.* (arrow from  $m$  to  $n$ )

- 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  $\rightarrow$  durchschn. #Kugeln pro Urne:  $\underline{\underline{n/m}}$



- Worst-case Laufzeit =  $\Theta(\max \#Kugeln \text{ pro Urne})$   
 mit hoher Wahrscheinlichkeit  $\in O\left(\frac{n}{m} + \frac{\log n}{\log \log n}\right)$   
 – bei  $n \leq m$  also  $O\left(\frac{\log n}{\log \log n}\right)$  falls  $m \geq n$  :  $O(1)$
- Erwartete Laufzeit (für jeden Schlüssel):  $f_{ind}(x)$   
 – Schlüssel in Tabelle: entspricht der  $\#Kugeln$  in der Urne einer zufälligen Kugel  
 – Schlüssel nicht in Tabelle:  $\#Kugeln$  einer zufälligen Urne  $\leftarrow O\left(1 + \frac{n}{m}\right)$

# Erwartete Laufzeit von Find

Load  $\alpha$  der Hashtabelle:

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

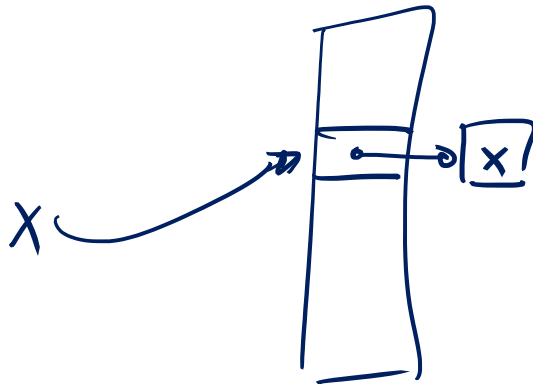
Kosten einer Suche:

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

$h(x)$  ist eine uniform zufällige Position

→ erwartete Listenlänge = durchschn. Listenlänge =  $\alpha$

Erwartete Laufzeit:



$$\frac{n-1}{m} \leq \alpha$$

$$\underline{\underline{1 + \alpha}}$$



## Load $\alpha$ der Hashtabelle:

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

## Kosten einer Suche:

- Suche nach einem Schlüssel  $x$ , welcher in der Hashtabelle ist  
Wieviele Schlüssel  $y \neq x$  sind in der Liste von  $x$ ?
- Die anderen Schlüssel sind zufällig verteilt, also entspricht die erwartete Anzahl  $y \neq x$  der erwarteten Länge einer zufälligen Liste in einer Hashtabelle mit  $n - 1$  Einträgen.
- Das sind  $\frac{n-1}{m} < \frac{n}{m} = \underline{\alpha} \rightarrow$  Erw. Listenlänge von  $x < \underline{1 + \alpha}$

Erwartete Laufzeit:  $O(1 + 1 + \alpha) = O(1 + \alpha)$

Zusammenfassung Laufzeiten:

**create & insert:**

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

**find & delete:**

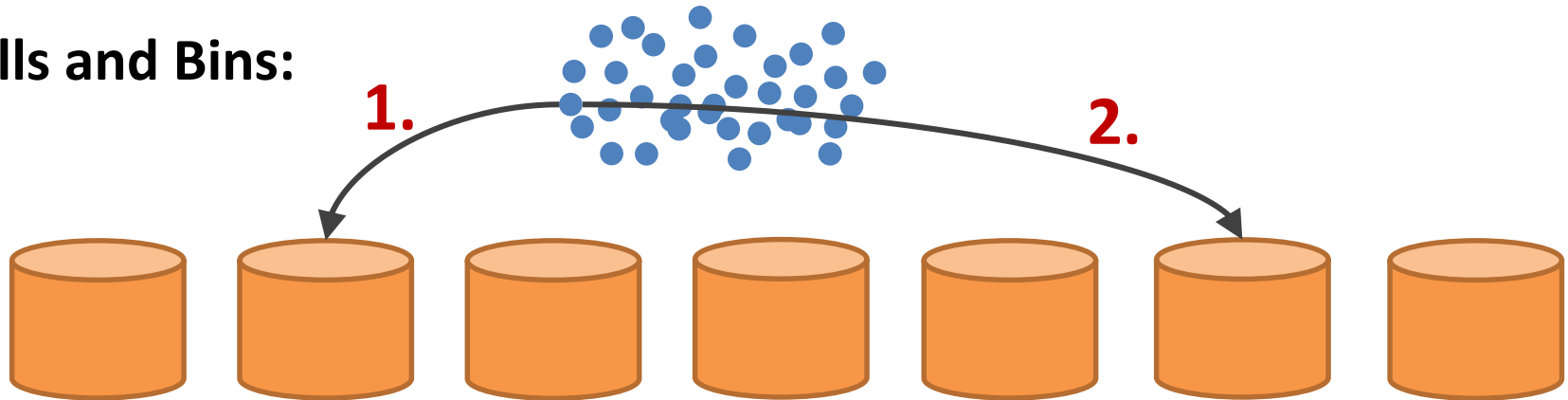
- Worst Case:  $\Theta(n)$
- Worst Case mit hoher Wahrsch. (bei zufälligem  $h$ ):  $O\left(\alpha + \frac{\log n}{\log \log n}\right)$
- Erwartete Laufzeit (für bestimmten Schlüssel  $x$ ):  $O(1 + \alpha)$ 
  - gilt für erfolgreiche und nicht erfolgreiche Suchen
  - Falls  $\alpha = O(1)$  (d.h., Hashtabelle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

# Kürzere Listenlängen

## Idee:

- Benutze zwei Hashfunktionen  $\underline{h_1}$  und  $\underline{h_2}$
- 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):

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

$$\underline{\underline{n \approx m}}$$

- Bekannt als “power of two choices”

**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



Wähle Hashfunktion als

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

0 .. m-1

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

## Vorteile:

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

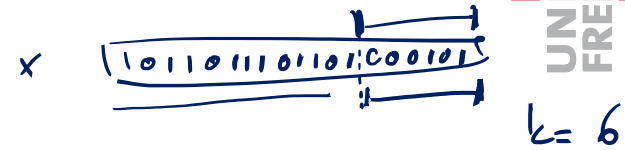
## Bemerkung:

- 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
- Am besten wählt man  $m$  als Primzahl
- Eine Primzahl  $m$ , so dass  $m = 2^k - 1$  ist auch keine gute Idee
  - (siehe Übungsblatt 4)
- Am besten: Primzahl  $m$ , welche nicht nahe bei einer 2er-Potenz ist

# Multiplikationsmethode

$\gamma - \text{LyJ}$

Wähle Hashfunktion als

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

$\in [0,1]$

$$0 \leq x < 2^w$$
$$0 \leq s < 2^w$$

$$m = 2^k$$

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

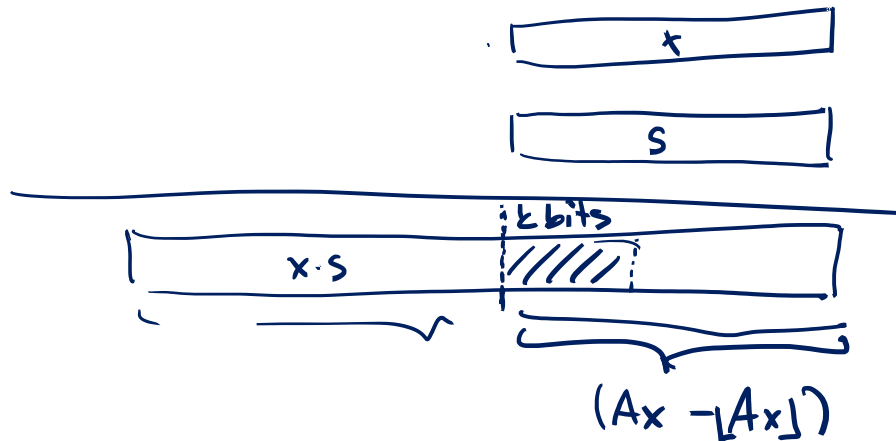
- $A$  ist eine Konstante zwischen 0 und 1

## 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

$$s \cdot x = A \cdot x \cdot 2^w$$

$$A = s \cdot 2^{-w}$$



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

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



Falls  $h$  zufällig aus allen möglichen Funktionen ausgewählt wird:

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

## 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  $\underline{h(x) = [m \cdot (Ax - [Ax])]}$  einfach den Parameter  $A$  zufällig wählen  $\uparrow$   
 $A \in \{0,1\}$
- Nicht ganz so gut, wie eine uniform zufällige Funktion, aber wenn man's richtig macht, funktioniert die Idee  $\rightarrow$  universelles Hashing

## Definition:

- Sei  $S$  die Menge der mögl. Schlüssel und  $m$  die Grösse der Hashtab.
- Sei  $\mathcal{H}$  eine Menge von Hashfunktionen  $S \rightarrow \{0, \dots, m - 1\}$
- Die Menge  $\mathcal{H}$  heisst  $c$ -universell, falls  $\text{Zuf. } h: \Pr(h(x) = h(y)) = \frac{1}{m}$  | $\mathcal{H}$ |

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

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

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

## Theorem:

- Sei  $\mathcal{H}$  eine  $c$ -universelle Menge von Hashfkt.  $S \rightarrow \{0, \dots, m - 1\}$
- 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  $\alpha$  Länge der Liste von  $x$

$$B_x := \{y \in X : h(y) = h(x)\} \quad 1 + \alpha$$

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

## Konsequenz:

- Im Erwartungswert sind alle Listen kurz!

# Universelles Hashing III

- Gute universelle Mengen von Hashfunktionen existieren!

## Beispiele:

$$M = m^{O(1)}$$

- $m$  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  $a, b \in S$

$C$ -universell für  $C \approx 1$

$$a = a_{k-1} a_{k-2} \dots a_0$$

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

$$124 = 1 \cdot 10^2 + 2 \cdot 10^1 + 4 \cdot 10^0$$

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.  $h_a(x) = (\sum_{i=0}^{k-1} a_i \cdot x_i) \bmod m$

$1$ -universell

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

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

- 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  
 $h(x,0)$ ,  $h(x,1)$ ,  $h(x,2)$ , ...,  $h(x,m-1)$

# Lineares Sondieren

## Idee:

$$h(x)$$

$$h(x, i) = h(x)$$

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

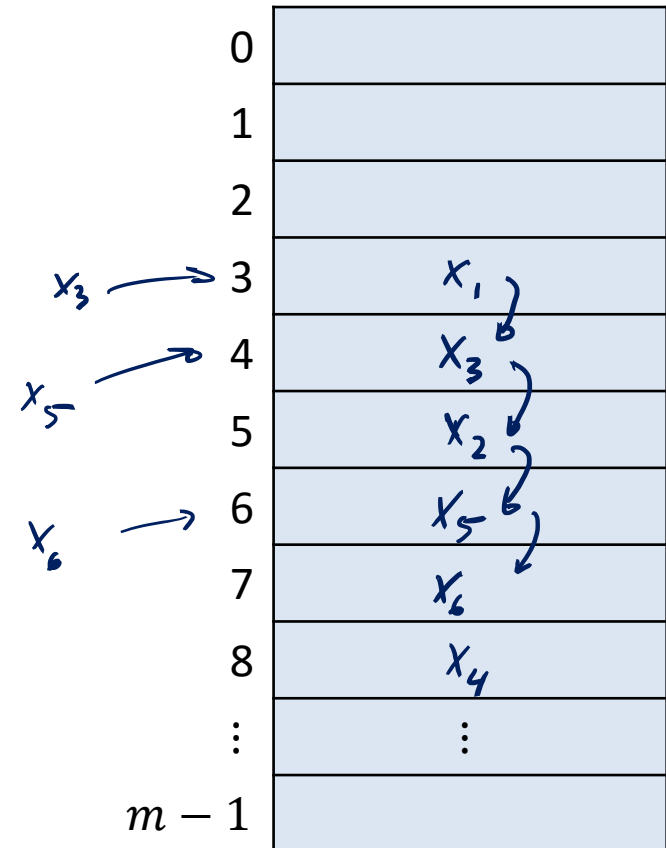
$$h(x, i) = (h(x) + i) \bmod m$$

für  $i = 0, \dots, m - 1$

## Beispiel:

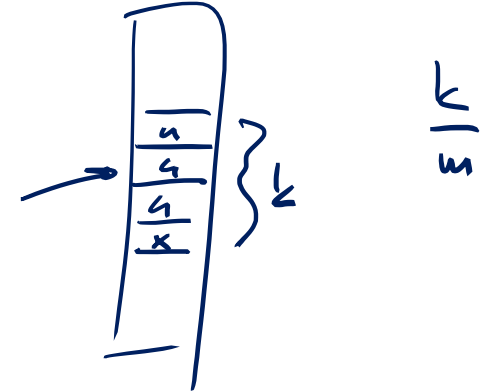
Füge folgende Schlüssel ein

- $x_1, h(x_1) = 3$
- $x_2, h(x_2) = 5$
- $x_3, h(x_3) = 3$
- $x_4, h(x_4) = 8$
- $x_5, h(x_5) = 4$
- $x_6, h(x_6) = 6$
- ...



## 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!!

## Idee:

- Nehme Sequenz, welche nicht zu Cluster führt:

$$h(x, i) = (h(x) + \underline{c_1}i + \underline{c_2}i^2) \bmod m$$

für  $i = 0, \dots, m - 1$

$$h(x) = h(y)$$

## 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  $h_1(x) = h_1(y)$

$$\underline{h(x, i)} = (\underline{h_1(x)} + i \cdot \underline{h_2(x)}) \bmod m$$

## Vorteile:

- Sondierfunktion 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:

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

- Funktioniert in der Praxis sehr gut!