

# Algorithmen und Datenstrukturen

## Vorlesung 4

### Dictionaries 1: Binäre Suche, Hashing mit Chaining & Offene Adressierung

Fabian Kuhn

Algorithmen und Komplexität



**UNI  
FREIBURG**

**Dictionary:** (auch: Maps, assoziative Arrays, Symbol Table)

- Verwaltet eine Kollektion von Elementen, wo bei jedes Element durch einen eindeutigen Schlüssel (key) repräsentiert wird

## Operationen:

- *create* : erzeugt einen leeren Dictionary
- *D.insert(key, value)* : fügt neues (*key,value*)-Paar hinzu
  - falls schon ein Eintrag für *key* besteht, wird er ersetzt
- *D.find(key)* : gibt Eintrag zu Schlüssel *key* zurück
  - falls ein Eintrag vorhanden (gibt sonst einen Default-Wert zurück)
- *D.delete(key)* : löscht Eintrag zu Schlüssel *key*

- Wir kümmern uns in einer ersten Phase nur um die Basisoperationen *insert*, *find*, *delete* (und *create*)

## Dictionary Beispiele:

- Wörterbuch (key: Wort, value: Definition / Übersetzung)
- Telefonbuch (key: Name, value: Telefonnummer)
- DNS Server (key: URL, value: IP-Adresse)
- Python Interpreter (key: Variablenname, value: Wert der Variable)
- Java/C++ Compiler (key: Variablenname, value: Typinformation)

**In all diesen Fällen ist insbesondere eine schnelle *find*-Op. wichtig!**

## Operationen:

- *create*:
  - lege neue leere Liste an
- *D.insert(key, value)*:
  - füge neues Element vorne ein
  - Annahme: Es gibt noch keinen Eintrag mit dem Schlüssel *key*
- *D.find(key)*:
  - gehe von vorne durch die Liste
- *D.delete(key)*:
  - suche zuerst das Listenelement (wie in *find*)
  - lösche Element dann aus der Liste
  - Bei einfach verketteten Listen muss man stoppen, sobald *current.next.key == key* ist!

## Laufzeiten:

*create*:  $O(1)$

*insert*:  $O(1)$

- Falls man nicht überprüfen muss, ob der Schlüssel schon vorkommt

*find*:  $O(n)$

- Wir müssen möglicherweise über die ganze Liste iterieren

*delete*:  $O(n)$

- Wir müssen möglicherweise über die ganze Liste iterieren

Ist das gut?

- Insbesondere find ist sehr teuer!

## Operationen:

- *create*:
  - lege neues Array der Länge  $NMAX$  an
- *D.insert(key, value)*:
  - füge neues Element hinten an (falls es noch Platz hat)
  - Annahme: Es gibt noch keinen Eintrag mit dem Schlüssel *key*
- *D.find(key)*:
  - gehe von vorne (oder hinten) durch die Elemente
- *D.delete(key)*:
  - suche zuerst nach dem *key*
  - lösche Element dann aus dem Array:

**Man muss alles dahinter um eins nach vorne schieben!**

## Laufzeiten:

*create:  $O(1)$*

*insert:  $O(1)$*

*find:  $O(n)$*

- Wir müssen möglicherweise über das ganze Array iterieren

*delete:  $O(n)$*

- Wir müssen möglicherweise über das ganze Array iterieren und im Worst Case  $\Omega(n)$  Werte umkopieren

## Bessere Ideen?

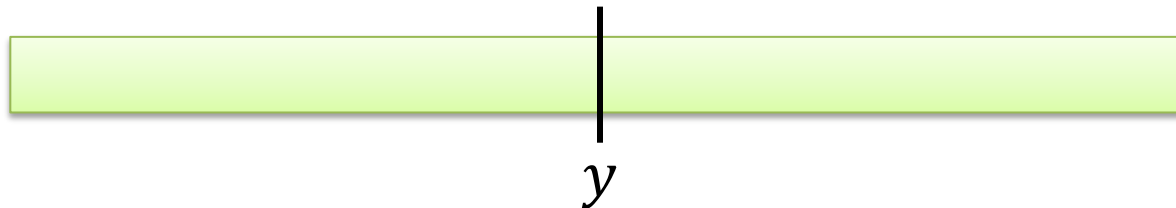
- Insbesondere find ist immer noch sehr teuer!

# Benutze sortiertes Array?

- **Teure Operation** bei Liste/Array, insbesondere *find*
- Falls (sobald) sich die Einträge nicht zu sehr ändern, ist *find* die wichtigste Operation!
- Kann man in einem (nach Schlüsseln) sortierten Array schneller nach einem bestimmten Schlüssel suchen?
  - Beispiel: Suche Tel.-Nr. einer Person im Telefonbuch...

## Ideen für Suche nach $x$ :

- Wir schlagen Telefonbuch mal ungefähr in der Mitte auf und schauen, ob der Name in der ersten oder in der zweiten Hälfte ist.



Ist  $y < x$  oder ist  $y > x$  oder ist  $y = x$ ?



# Binäre Suche

Benutze Divide and Conquer Idee!

Suche nach der Zahl (dem Key) 19:

2	3	4	6	9	12	15	16	17	18	19	20	24	27	29
---	---	---	---	---	----	----	----	----	----	----	----	----	----	----

2	3	4	6	9	12	15	16	17	18	19	20	24	27	29
---	---	---	---	---	----	----	----	----	----	----	----	----	----	----

## Algorithmus (Array $A$ der Länge $n$ , Suche nach Schlüssel $x$ ):

- Behalte linken und rechten Rand  $l$  und  $r$ , so dass (falls  $x$  in  $A$  ist)

$$A[l] \leq x \leq A[r]$$

- Am Anfang setzen wir  $l = 0$  und  $r = n - 1$
- Gehe in die Mitte  $m = (l + r)/2$ 
  - Falls  $A[m] = x \Rightarrow x$  gefunden!
  - Falls  $A[m] < x \Rightarrow x$  ist im rechten Teil  $\Rightarrow l = m + 1$
  - Falls  $A[m] > x \Rightarrow x$  ist im linken Teil  $\Rightarrow l = m - 1$

2	3	4	6	9	12	15	16	17	18	19	20	24	27	29
---	---	---	---	---	----	----	----	----	----	----	----	----	----	----

**Algorithmus (Array  $A$  der Länge  $n$ , Suche nach Schlüssel  $x$ ):**

```
l = 0; r = n - 1;
while r > l do
    m = (l + r) / 2;
    if A[m] < x then
        l = m + 1
    else if A[m] > x then
        r = m - 1
    else
        l = m; r = m
```

Falls Schlüssel  $x$  im Array ist, dann gilt am Schluss  $A[l] = x$

## Wie überprüft man das?

- Empirisch: Unit Test oder auch systematischere Tests...
- **Formal?**
  - Korrektheit ist (meistens) noch wichtiger als Performance!

## Hoare Kalkül

- Wir schauen hier nur die Grundideen an
- **Vorbedingung**
  - Bedingung, welche am Anfang (der Methode / Schleife / ...) gilt
- **Nachbedingung**
  - Bedingung, welche am Schluss (der Methode / Schleife / ...) gilt
- **Schleifeninvariante**
  - Bedingung welche am Anfang / Ende jedes Schleifendurchlaufs gilt

# Ist der Algorithmus korrekt?

```
l = 0; r = n - 1;
while r > l do
    m = (l + r) / 2;
    if A[m] < x then l = m + 1
    else if A[m] > x then r = m - 1
    else l = m; r = m
```

## Vorbedingung

- *Array ist am Anfang sortiert, Array hat Länge  $n$*

## Nachbedingung

- *Falls  $x$  im Array ist, dann gilt  $A[l] = x$*

## Schleifeninvariante

- *Falls  $x$  im Array ist, dann gilt  $A[l] \leq x \leq A[r]$*

# Ist der Algorithmus korrekt?

## Vorbedingung

- *Array ist am Anfang sortiert, Array hat Länge  $n$*

$l = 0; r = n - 1;$

Schleifeninvariante

## Schleifeninvariante

- *Falls  $x$  im Array ist, dann gilt  $A[l] \leq x \leq A[r]$*
- Vorbedingung und Zuweisung zu  $l$  und  $r \rightarrow$  Schleifeninvariante
  - Invariante gilt am Anfang des ersten Schleifendurchlaufs

## Nachbedingung

- *Falls  $x$  im Array ist, dann gilt  $A[l] = x$*
- Abbruchbedingung while-Schleife  $\rightarrow l \geq r$  und damit  $A[l] \geq A[r]$
- Falls  $x$  im Array ist, dann folgt aus der Schleifeninvariante und da  $A$  sortiert ist, dass  $A[l] = A[r]$  und damit  $A[l] = x$

# Ist der Algorithmus korrekt?

```
l = 0; r = n - 1;
```

```
while r > l do
```

```
    m = (l + r) / 2;
```

```
    if A[m] < x then l = m + 1
```

```
    else if A[m] > x then r = m - 1
```

```
    else l = m; r = m
```

## Schleifeninvariante

- *Falls  $x$  im Array ist, dann gilt  $A[l] \leq x \leq A[r]$* 
  - Die Schleifeninvariante gilt am Anfang der Schleife, sie kann nur ungültig werden, wenn wir die Variablen  $l$  und  $r$  verändern
  - Wenn wir  $l = m + 1$  setzen, dann wissen wir, dass  $A[m] < x$ , daher gilt danach  $A[m + 1] \leq x$  falls  $x$  enthalten ist.
  - Analog, wenn wir  $r = m - 1$  setzen, dann wissen wir, dass  $A[m] > x$ , daher gilt danach  $x \leq A[m - 1]$  falls  $x$  enthalten ist.

# Terminiert der Algorithmus?

```
l = 0; r = n - 1;
```

```
while r > l do
```

```
    m = (l + r) / 2;
```

```
    if A[m] < x then l = m + 1
```

```
    else if A[m] > x then r = m - 1
```

```
    else l = m; r = m
```

- Veränderung der Anz. Elemente ( $r - l + 1$ ) pro Schleifendurchlauf?

- $l = m + 1$ :

$$r - (m + 1) + 1 \leq r - \left(\frac{l + r}{2} + \frac{1}{2}\right) + 1 = \frac{r - l + 1}{2}$$

- $r = m - 1$ :

$$(m - 1) - l + 1 \leq \frac{l + r}{2} - 1 - l + 1 = \frac{r - l}{2} < \frac{r - l + 1}{2}$$

- Sonst wird  $x$  gefunden und  $r - l + 1$  wird 1



## Terminiert der Algorithmus?

- In jedem Schleifendurchlauf wird die Anzahl der Elemente mindestens halbiert.
- Der Algorithmus terminiert!

## Laufzeit?

$$T(n) \leq T(\lfloor n/2 \rfloor) + c, \quad T(1) \leq c$$

$$T(n) \leq T(n/2) + c$$

$$\leq T(n/4) + \underbrace{c + c}_{2c}$$

$$\leq T(n/8) + 3c$$

$$\vdots$$
$$\leq T(n/2^i) + i \cdot c$$

$$\leq T(1) + c \cdot (\log_2 n) \leq c(\log_2 n + 1)$$

# Laufzeit Binäre Suche

Der Algorithmus terminiert in Zeit  $O(\log n)$ .

$$T(n) \leq T(n/2) + c, \quad T(1) \leq c$$

Vermutung:  $T(n) \leq c(\log_2 n + 1)$

Verankerung:  $n=1 \quad T(1) \leq c(0 + 1) = c \quad \checkmark$

Schritt:  $n > 1 \quad T(n) \leq T(n/2) + c$

$$\leq c(\underbrace{\log_2 \frac{n}{2} + 1}_{\log_2 n}) + c$$
$$= c(\log_2 n + 1),$$

## Operationen:

- *create*:
  - lege neues Array der Länge *NMAX* an
- *D.find(key)*:
  - **Suche nach *key* mit binärer Suche**
- *D.insert(key, value)*:
  - suche nach *key* und füge neues Element an der richtigen Stelle ein
  - Einfügen: alles dahinter muss um eins nach hinten geschoben werden!
- *D.delete(key)*:
  - suche zuerst nach dem *key* und lösche den Eintrag
  - Löschen: alles dahinter muss um eins nach vorne geschoben werden!

## Laufzeiten:

*create*:  $O(1)$

*insert*:  $O(n)$

*find*:  $O(\log n)$

*delete*:  $O(n)$

Können wir alle Operationen schnell machen?

- und das *find* noch schneller?

# Dictionary bis jetzt

- Bis jetzt sahen wir 3 einfache Dictionary Implementierungen

	Verkettete Liste (unsortiert)	Array (unsortiert)	Array (sortiert)
insert	$O(1)$	$O(1)$	$O(n)$
delete	$O(n)$	$O(n)$	$O(n)$
find	$O(n)$	$O(n)$	$O(\log n)$

$n$ : Aktuelle Anzahl Elemente im Dictionary

- Wichtigste Operation oft: find
- Können wir das find noch weiter verbessern?
- Können wir alle Operationen schnell haben?

# 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	<b>Value 1</b>
3	None
4	None
5	None
6	<b>Marc</b>
7	<b>Value 3</b>
8	None
⋮	⋮
$M - 1$	None

*find(2) → "Value 1"*

*insert(6, "Marc")*

*delete(4)*

## 1. Direkte Adressierung benötigt zu viel Platz!

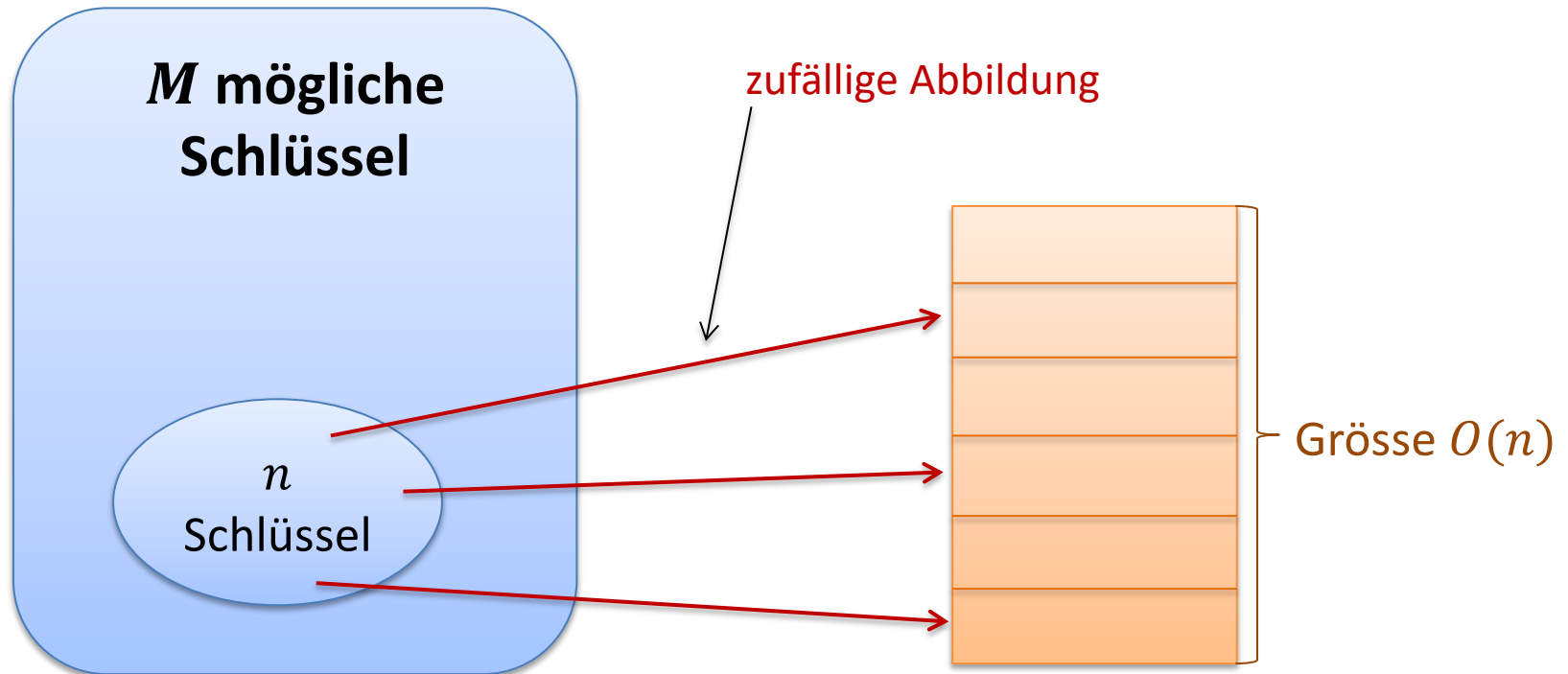
- Falls Schlüssel ein beliebiger *int* (32 bit) sein kann:  
Wir benötigen ein Array der Grösse  $2^{32} \approx 4 \cdot 10^9$ .  
Bei 64 bit Integers sind's sogar schon mehr als  $10^{19}$ ...

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

- Wo kommt das *(key,value)*-Paar ("*Philipp*", "*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!**

## 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)

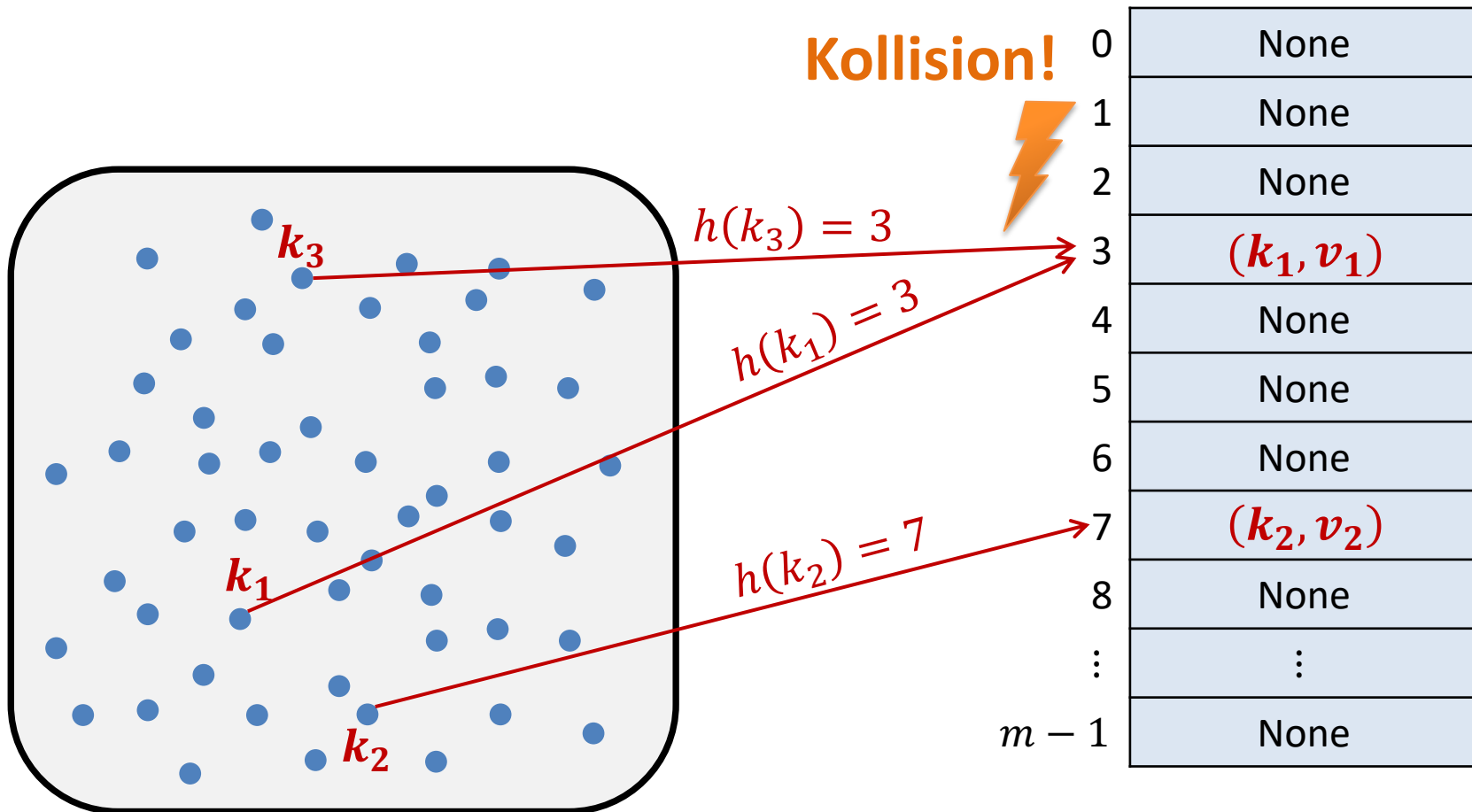
## Hashfunktion

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

- 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.  $insert(k_1, v_1)$
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?
  - Das ist nur möglich, wenn man die Menge der benutzten Schlüssel im Voraus kennt.
  - Selbst dann ist es unter Umständen sehr teuer, eine solche Hashfunktion zu finden.
- Eine andere Hashfunktion nehmen?
  - Man müsste dann bei jeder neuen Kollision wieder eine neue Hashfunktion wählen
  - Eine neue Hashfunktion heisst, dass man alle bestehenden Werte in der Hashtabelle umkopieren muss.
- Weitere Ideen?

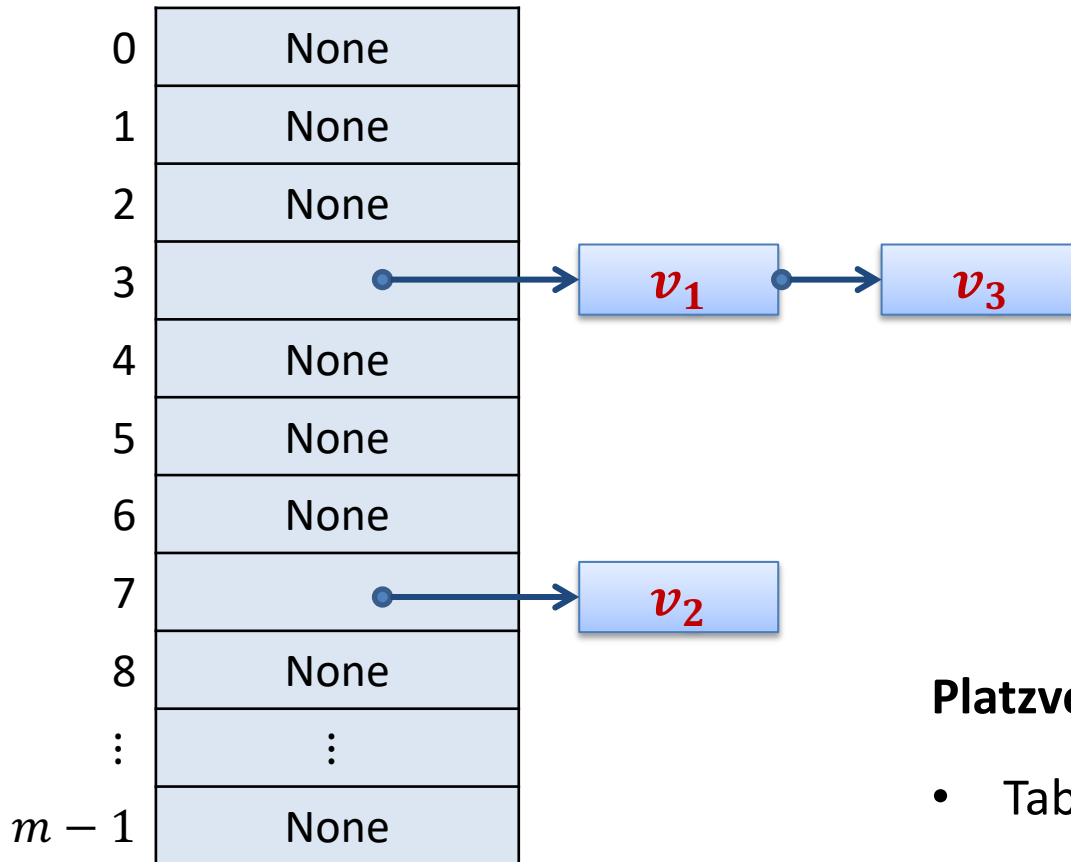
## Kollisionen Lösungsansätze

- 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

## Hashtabelle



Platzverbrauch:  $O(m + n)$

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

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

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

*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...

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

*create:*  $O(1)$

*insert:*  $O(1 + \text{Listenlänge})$

- Falls man nicht überprüfen muss, ob der Schlüssel schon vorkommt, dann sind die insert-Kosten sogar  $O(1)$ .

*find:*  $O(1 + \text{Listenlänge})$

*delete:*  $O(1 + \text{Listenlänge})$

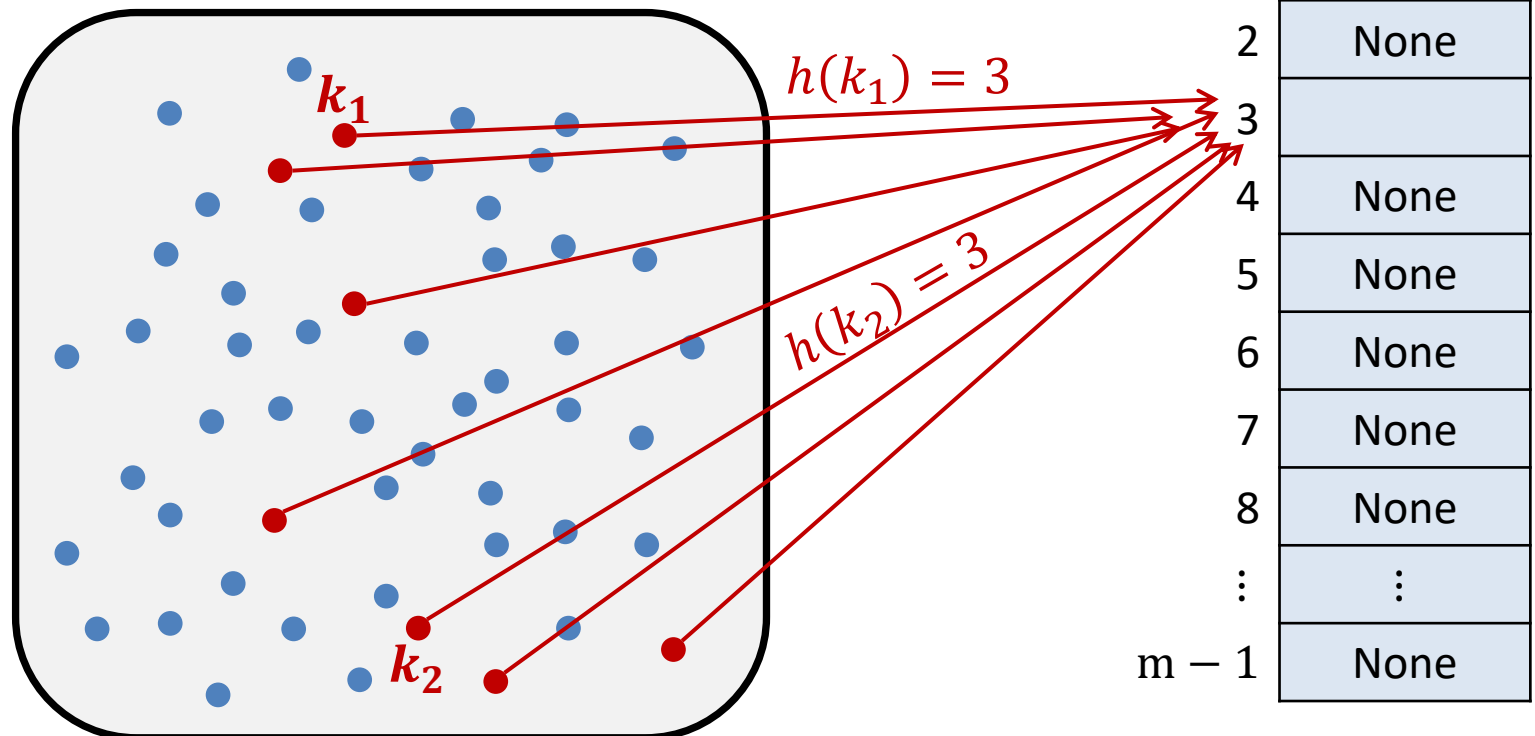
- Wir müssen also anschauen, wie lang die Listen werden.

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





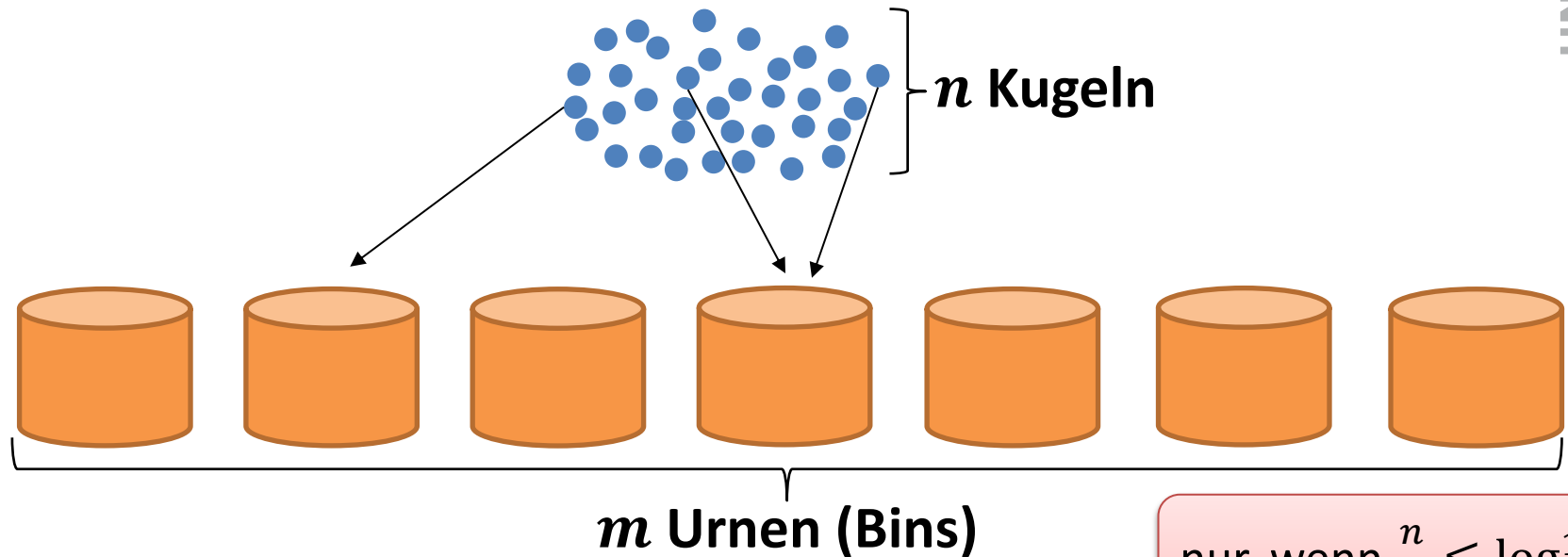
# Länge der verketteten Liste

- Kosten von *insert*, *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**

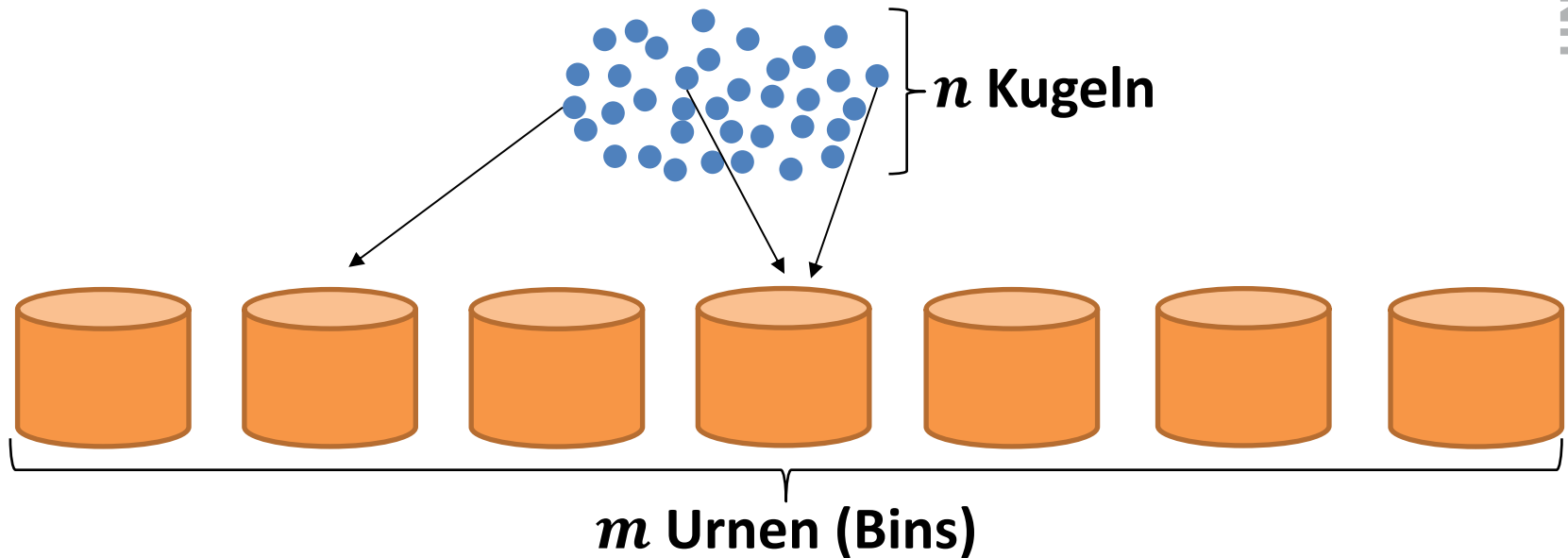
- Jede Kugel wird (unabhängig) in eine zufällige Urne geworfen
- Längste Liste = maximale Anz. Kugeln in der gleichen Urne
- Durchschnittliche Listenlänge = durchschn. Anz. Kugeln pro Urne

$m$  Urnen,  $n$  Kugeln  $\rightarrow$  durchschn. #Kugeln pro Urne:  $n/m$



nur, wenn  $\frac{n}{m} \leq \log^{1-\epsilon} n$

- Worst-case Laufzeit =  $\Theta(\max \#Kugeln \text{ pro Urne})$   
mit hoher Wahrscheinlichkeit (whp)  $\in O\left(\frac{n}{m} + \frac{\log n}{\log \log n}\right)$ 
  - bei  $n \leq m$  also  $O\left(\frac{\log n}{\log \log n}\right)$
- Die längste Liste wird also Länge  $\Theta\left(\frac{\log n}{\log \log n}\right)$  haben.



## Erwartete Laufzeit (für jeden Schlüssel):

- Schlüssel in Tabelle:
  - Liste eines zufälligen Eintrags
  - entspricht der #Kugeln in der Urne einer zufälligen Kugel
- Schlüssel nicht in Tabelle:
  - Länge einer zufälligen Liste, d.h. #Kugeln einer zufälligen Urne

# Erwartete Laufzeit von Find

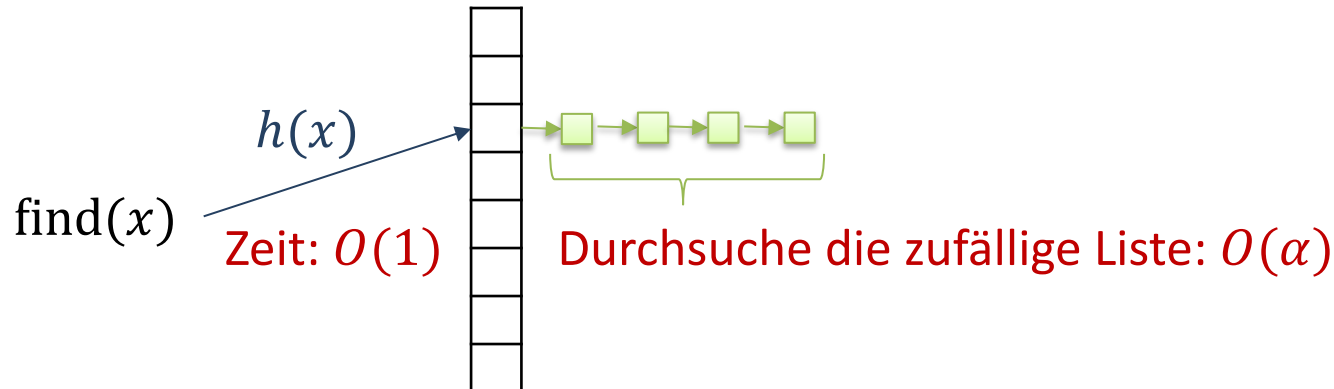
Load  $\alpha$  der Hashtabelle:

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

**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:  $O(1 + \alpha)$**



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  $\frac{n-1}{m} < \frac{n}{m} = \alpha \rightarrow$  Erw. Listenlänge von  $x < 1 + \alpha$

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

## create:

- Laufzeit  $O(1)$

## insert, 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.**

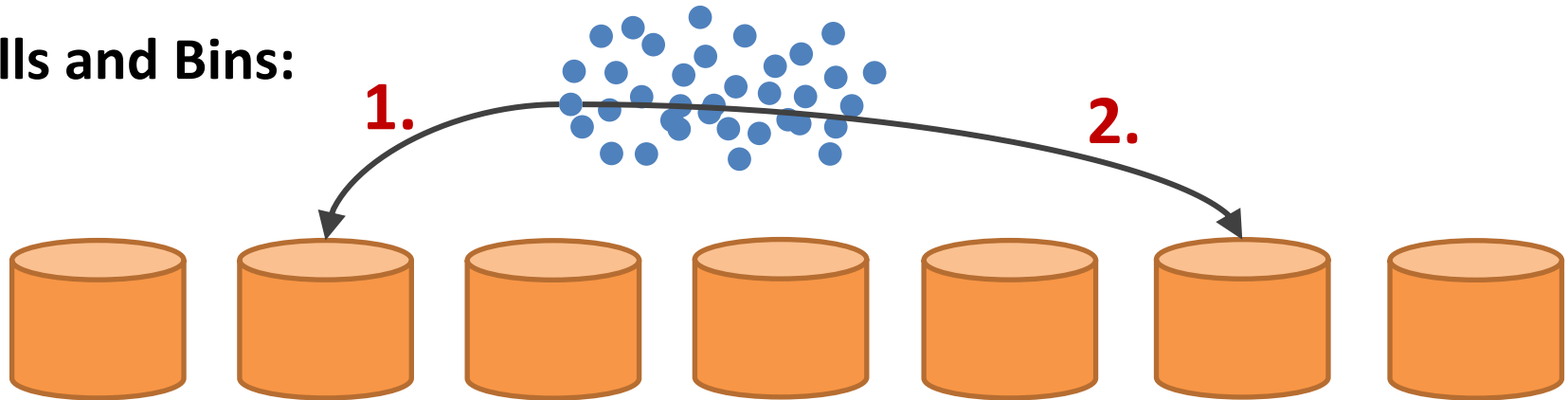
nur, wenn  $\alpha \leq \log^{1-\varepsilon} n$

# 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):  
$$n/m + O(\log \log m)$$
- Bekannt als “power of two choices”

## 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

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

- Ergibt Hashwerte  $h(x, 0), h(x, 1), h(x, 2), \dots, h(x, m - 1)$
- Für jedes  $x \in S$  sollte  $h(x, i)$  durch alle  $m$  Werte gehen (für versch.  $i$ )
- Einfügen eines Elements mit Schlüssel  $x$ :
  - Versuche der Reihe nach an den Positionen  
 $h(x, 0), h(x, 1), h(x, 2), \dots, h(x, m - 1)$



## Idee:

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

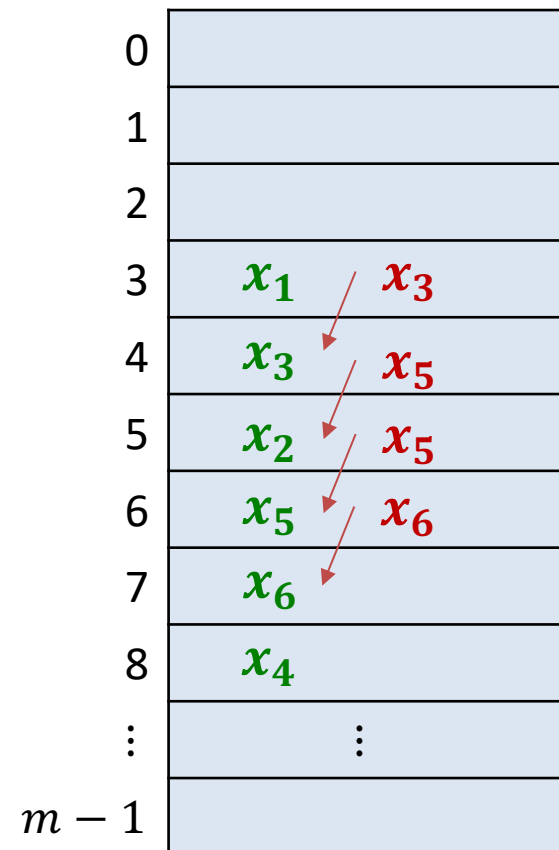
$$h(x, i) = (h(x) + i) \bmod m$$

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  $(k + 2)/m$
- Je grösser die Cluster, desto schneller wachsen sie!!

## Idee:

- Nehme Sequenz, welche nicht zu Cluster führt:

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

für  $i = 0, \dots, m - 1$

## Vorteil:

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

**Nachteil:**  $h(x) = h(y) \implies h(x, i) = h(y, i)$

- 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

**Idee:** Benutze zwei Hashfunktionen

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

**Vorteile:**

- Falls  $m$  eine Primzahl ist, werden alle Positionen abgedeckt
- 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:

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

- Funktioniert in der Praxis sehr gut!

## Offene Adressierung:

- Schlüssel  $x$  kann an folgenden Positionen sein:

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

## Operation Find?

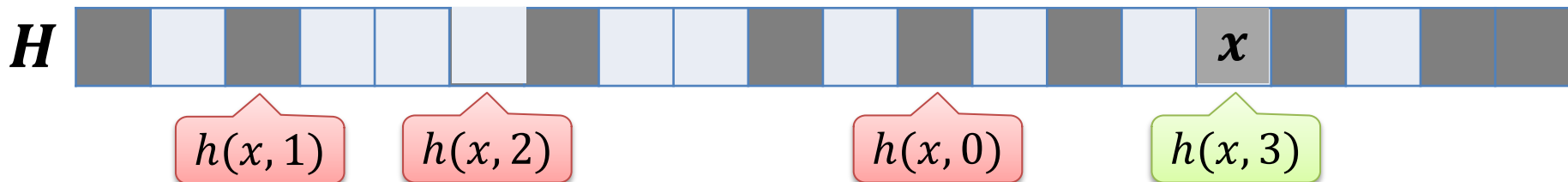
Hashtabelle

$i = 0$

```
while  $i < m$  and  $H[h(x, i)] \neq \text{None}$  and  $H[h(x, i)].\text{key} \neq x$ :  
     $i += 1$ 
```

```
return ( $i < m$  and  $H[h(x, i)] \neq \text{None}$ )
```

Beim Einfügen von  $x$  wird  $x$  an Stelle  $H[h(x, i)]$  eingefügt, wenn  $H[h(x, j)]$  für  $j < i$  besetzt ist



## Offene Adressierung:

- Schlüssel  $x$  kann an folgenden Positionen sein:

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

## Operation Delete

$i = 0$

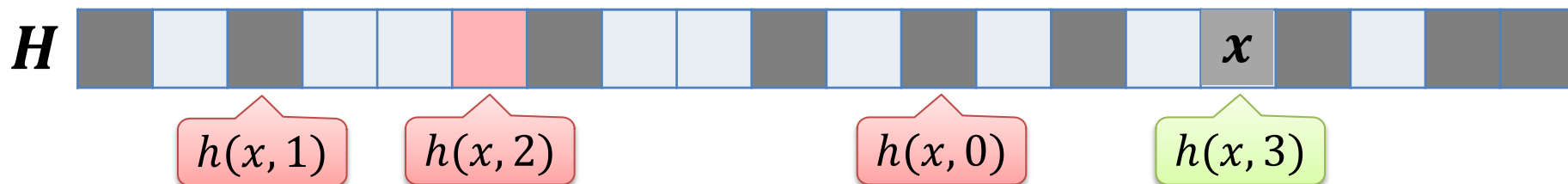
while  $i < m$  and  $H[h(x, i)] \neq \text{None}$  and  $H[h(x, i)].\text{key} \neq x$ :

$i += 1$

if  $i < m$  and  $H[h(x, i)] \neq \text{None}$ :

$H[h(x, i)] = \text{deleted}$

Beim Einfügen von  $x$  wird  $x$  an Stelle  $H[h(x, i)]$  eingefügt, wenn  $H[h(x, j)]$  für  $j < i$  besetzt ist



# Offene Adressierung: Operation Find

## Offene Adressierung:

- Schlüssel  $x$  kann an folgenden Positionen sein:

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

## Operation Find

$i = 0$

```
while  $i < m$  and  $H[h(x, i)] \neq \text{None}$  and  $H[h(x, i)].\text{key} \neq x$ :  
     $i += 1$ 
```

```
return ( $i < m$  and  $H[h(x, i)] \neq \text{None}$ )
```

Beim Einfügen von  $x$  wird  $x$  an Stelle  $H[h(x, i)]$  eingefügt, wenn  $H[h(x, j)]$  für  $j < i$  besetzt ist



## Offene Adressierung:

- Alle Schlüssel/Werte werden direkt im Array gespeichert
  - Gelöschte Einträge müssen markiert werden
- Keine Listen nötig
  - spart den dazugehörigen Overhead...
- Nur schnell, solange der Load

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

nicht zu gross wird...

- dann ist's dafür in der Praxis besser als Chaining...
- $\alpha > 1$  ist nicht möglich!
  - da nur  $m$  Positionen zur Verfügung stehen



## 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 noch 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.

Hashtabellen (Dictionary):

<https://docs.python.org/2/library/stdtypes.html#mapping-types-dict>

- neue Tabelle generieren: `table = {}`
- $(key, value)$ -Paar einfügen: `table.update({key : value})`
- Suchen nach *key*:  
`key in table`  
`table.get(key)`  
`table.get(key, default_value)`
- Löschen von *key*:  
`del table[key]`  
`table.pop(key, default_value)`

## Java-Klasse HashMap:

- Neue Hashtab. erzeugen (Schlüssel vom Typ  $K$ , Werte vom Typ  $V$ )  
`HashMap<K,V> table = new HashMap<K,V>();`
- Einfügen von  $(key,value)$ -Paar ( $key$  vom Typ  $K$ ,  $value$  vom Typ  $V$ )  
`table.put(key, value)`
- Suchen nach  $key$   
`table.get(key)`  
`table.containsKey(key)`
- Löschen von  $key$   
`table.remove(key)`
- Ähnliche Klasse HashSet: verwaltet nur Menge von Schlüssel

Es gibt nicht eine Standard-Klasse

## **hash\_map:**

- Sollte bei fast allen C++-Compilern vorhanden sein

[http://www.sgi.com/tech/stl/hash\\_map.html](http://www.sgi.com/tech/stl/hash_map.html)

## **unordered\_map:**

- Seit C++11 in Standard STL

[http://www.cplusplus.com/reference/unordered\\_map/unordered\\_map/](http://www.cplusplus.com/reference/unordered_map/unordered_map/)

## C++-Klassen `hash_map` / `unordered_map`:

- Neue Hashtab. erzeugen (Schlüssel vom Typ  $K$ , Werte vom Typ  $V$ )  
`unordered_map<K,V> table;`
- Einfügen von  $(key,value)$ -Paar ( $key$  vom Typ  $K$ ,  $value$  vom Typ  $V$ )  
`table.insert(key, value)`
- Suchen nach  $key$   
`table[key]` oder `table.at(key)`  
`table.count(key) > 0`
- Löschen von  $key$   
`table.erase(key)`

## Achtung

- Man kann eine `hash_map` / `unordered_map` in C++ wie ein Array benutzen
  - *die Array-Elemente sind die Schlüssel*

- Aber:

`T[key]` fügt den Schlüssel *key* ein, falls er noch nicht drin ist

`T.at(key)` wirft eine Exception falls *key* nicht in der Map ist