# Algorithmen und Datenstrukturen

## Vorlesung 4

## Hash Tabellen 1:
## Hashing mit Chaining & Offene Adressierung

Fabian Kuhn
Algorithmen und Komplexität

UNI FREIBURG

# Abstrakte Datentypen : Dictionary

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

- Verwaltet eine Menge 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*

# Dictionary bis jetzt

- Bis jetzt sahen wir 3 einfache Dictionary Implementierungen

| | **Verkettete Liste (unsortiert)** | **Array (unsortiert)** | **Array (sortiert)** |
|---|---|---|---|
| **insert** | $\boldsymbol{O(1)}$ | $\boldsymbol{O(1)}$ | $\boldsymbol{O(n)}$ |
| **delete** | $\boldsymbol{O(n)}$ | $\boldsymbol{O(n)}$ | $\boldsymbol{O(n)}$ |
| **find** | $\boldsymbol{O(n)}$ | $\boldsymbol{O(n)}$ | $\boldsymbol{O(\log n)}$ |

$n$: 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 | **Value 1** |
| 3 | None |
| 4 | None |
| 5 | None |
| 6 | **Philipp** |
| 7 | **Value 3** |
| 8 | None |
| $\vdots$ | $\vdots$ |
| $M - 1$ | None |

*find(2) → "Value 1"*

*insert(6, "Philipp")*

*delete(4)*

# Direkte Adressierung : Probleme

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?

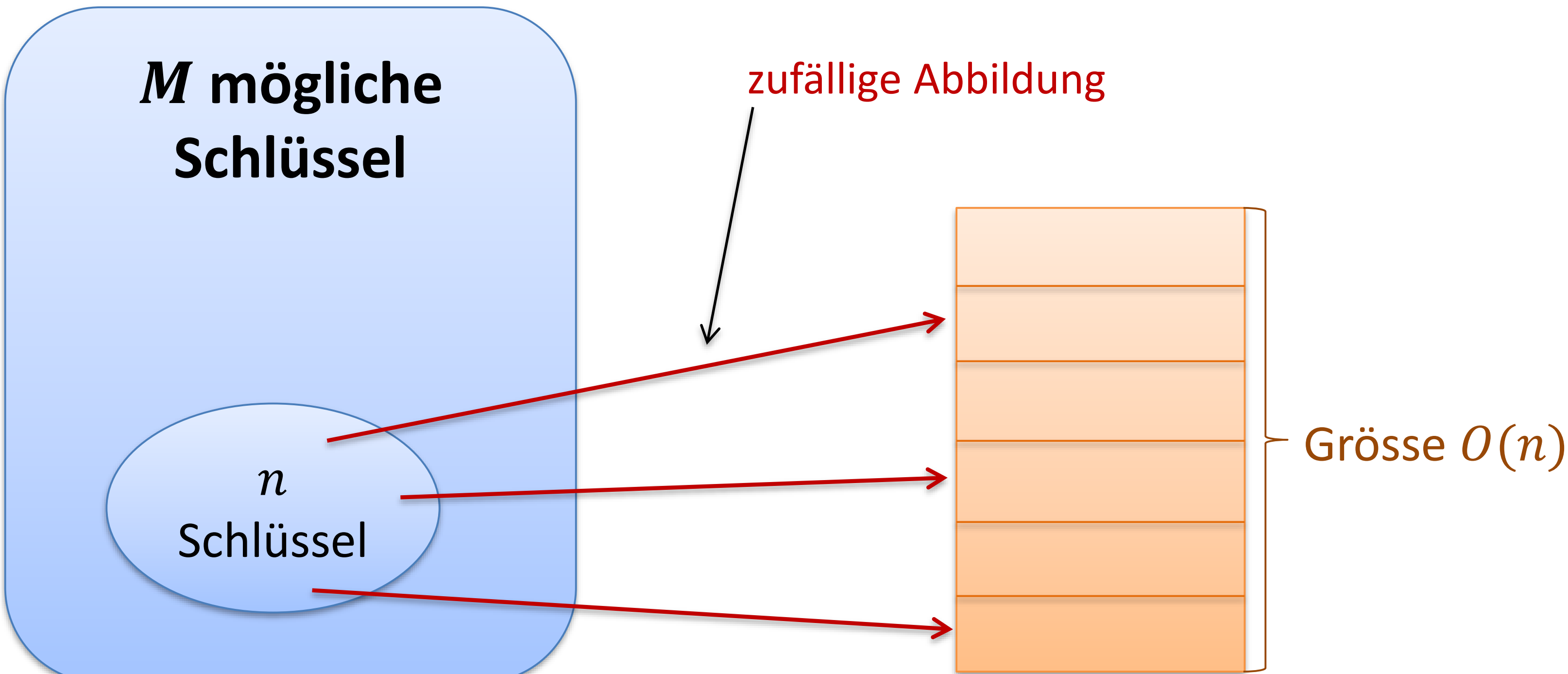
   – Pythagoras: "Alles ist Zahl"
   "Alles" kann als Folge von Bits abgespeichert werden:
   **Interpretiere Bit-Folge als ganze Zahl**

   – **Verschärft das Platz-Problem noch zusätzlich!**

# 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

**Schlüsselraum $\boldsymbol{S}$, $|\boldsymbol{S}| = \boldsymbol{M}$** (alle möglichen Schlüssel)

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

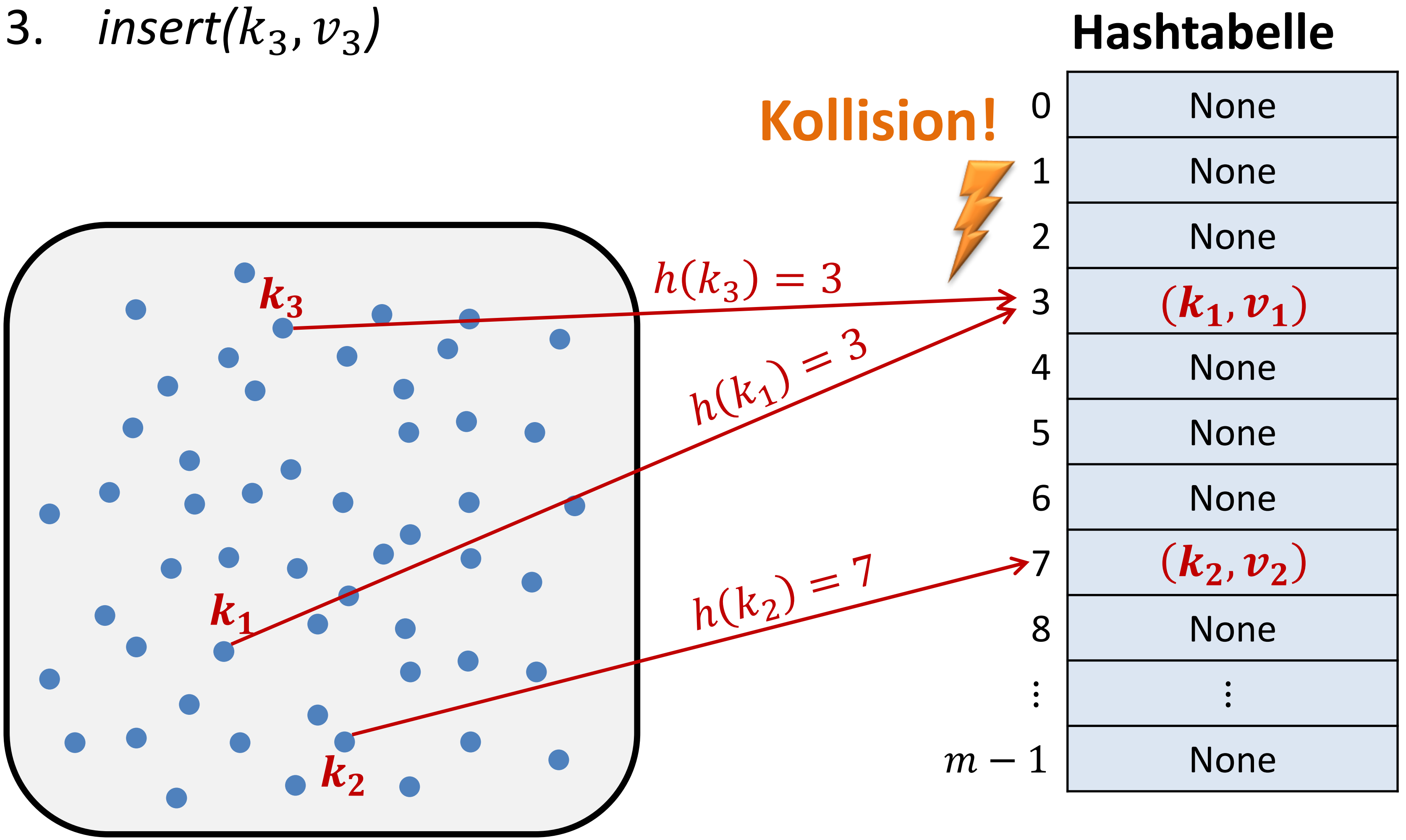
**Hashfunktion**

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

- 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*($k_3, v_3$)



# Hashtabellen : Kollisionen

**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?

# Hashtabellen : Kollisionen

## Kollisionen Lösungsansätze

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

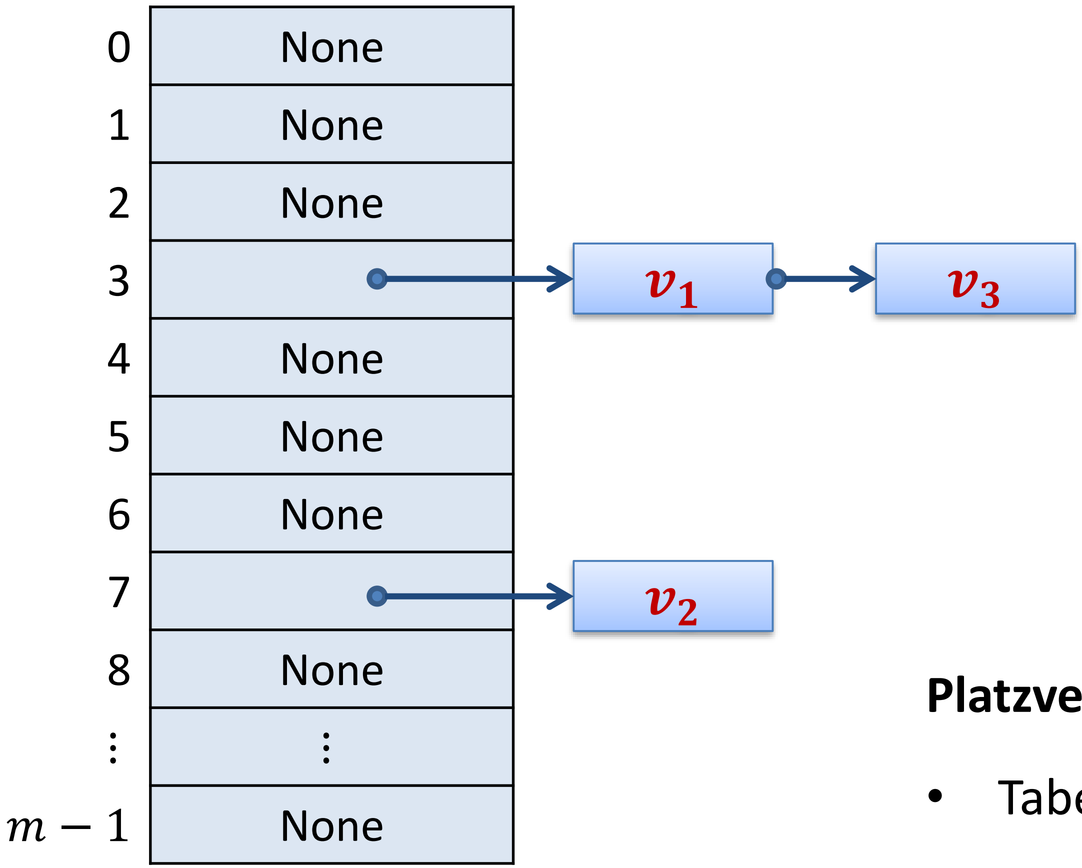
1. Speichere beide (key,value)-Paare an die gleiche Stelle
   - Die Hashtabelle muss an jeder Position Platz für mehrere Elemente bieten
   - Wir wollen die Hashtabelle aber nicht einfach vergrössern
     (dann könnten wir gleich mit einer grösseren Tabelle starten...)
   - **Lösung: Verwende verkettete 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 verkette Liste

**Hashtabelle**




**Platzverbrauch:** $\boldsymbol{O(m+n)}$

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

# Laufzeit Hashtabellen-Operationen

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

*create:* $\boldsymbol{O(1)}$

*insert:* $\boldsymbol{O(1)}$

*find:* $\boldsymbol{O(1)}$

*delete:* $\boldsymbol{O(1)}$

- 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

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

*create:* $\boldsymbol{O}(1)$

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

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

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

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

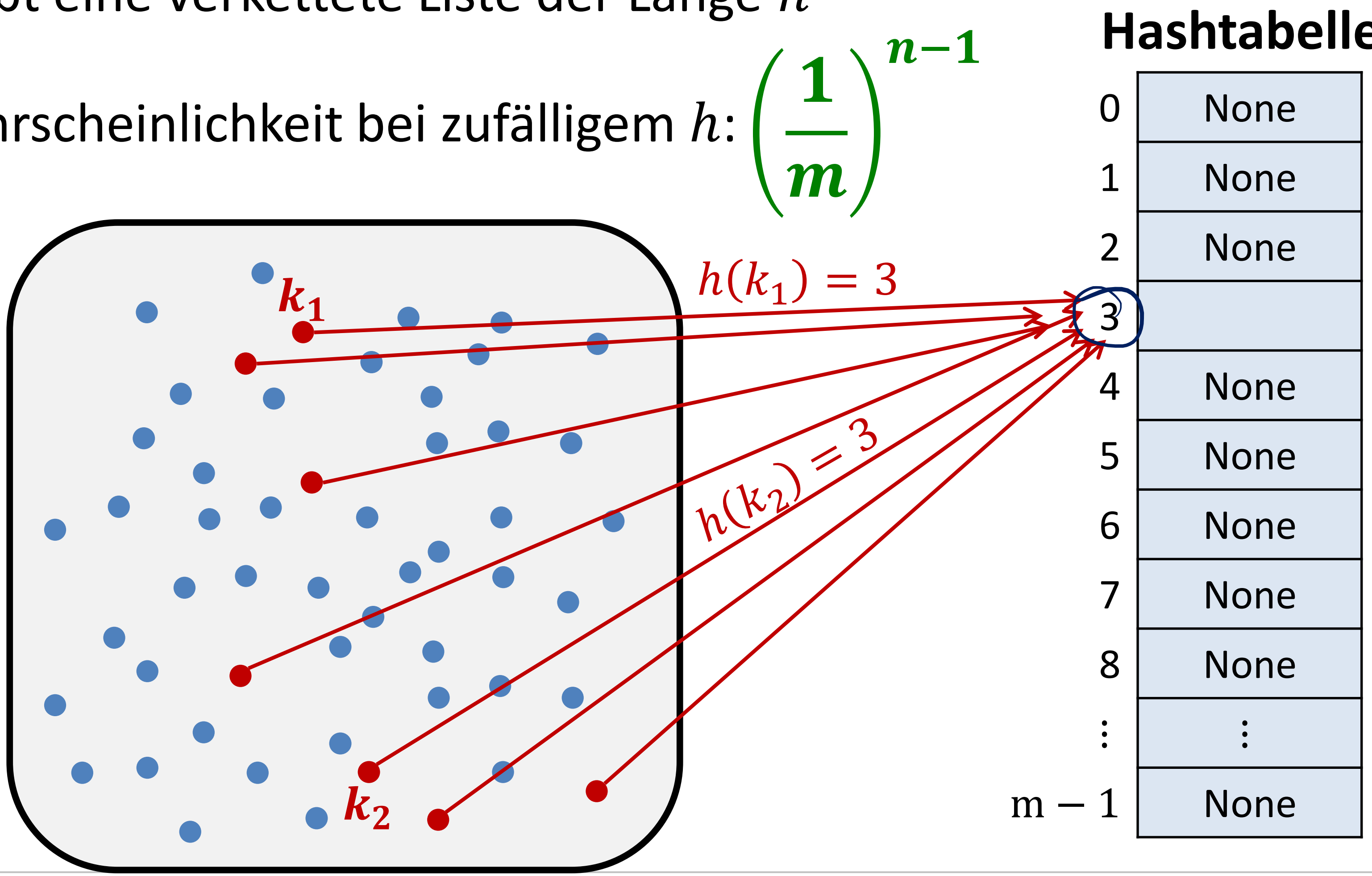
- Wir müssen also anschauen, wie lang die Listen werden.

# Funktionsweise Hashtabellen

m: Größe der Tabelle
n: # Schlüssel

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 verketten Liste

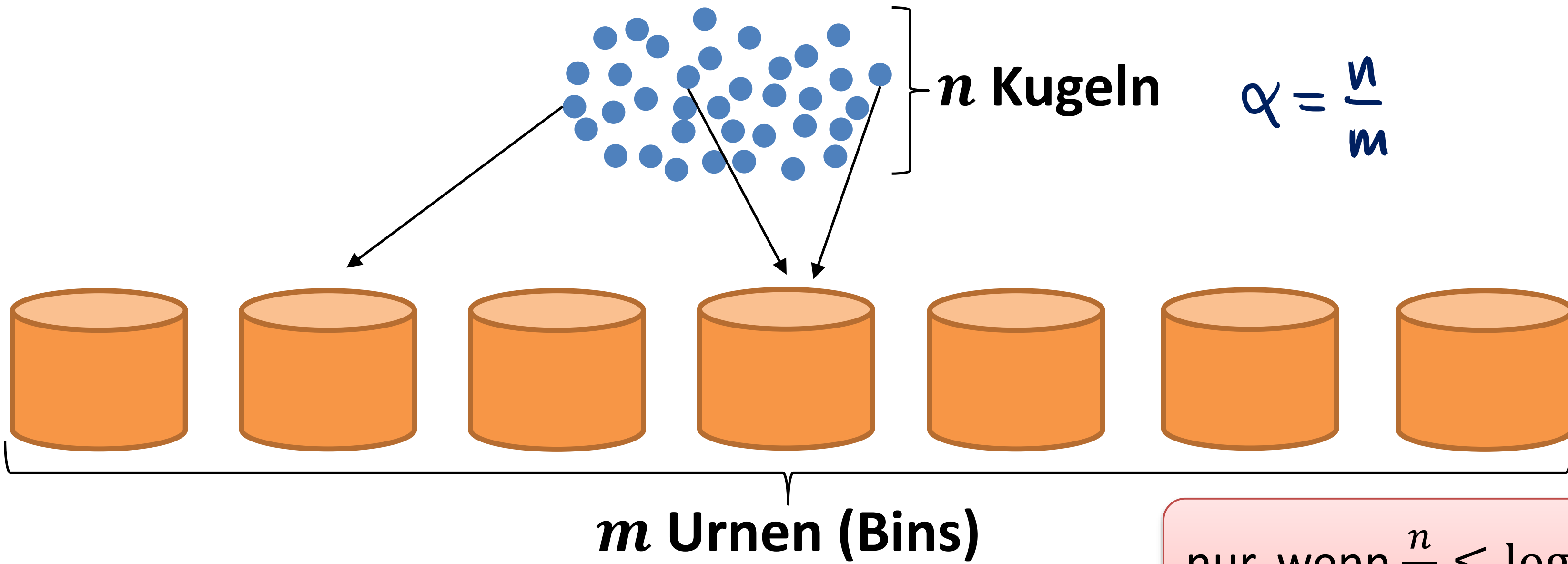
- Kosten von *insert*, *find* und *delete* hängt von der Länge der entprechenden Liste ab
- Wie lang werden die Listen?
  - Annahme: Grösse der Hashtabelle $m$, Anzahl Elemente $n$
  - Weitere Annahme: Hashfunktion $h$ verhält sich wie zufällige Funktion
- Listenlängen entspricht folgendem Zufallsexperiment

**$m$ 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



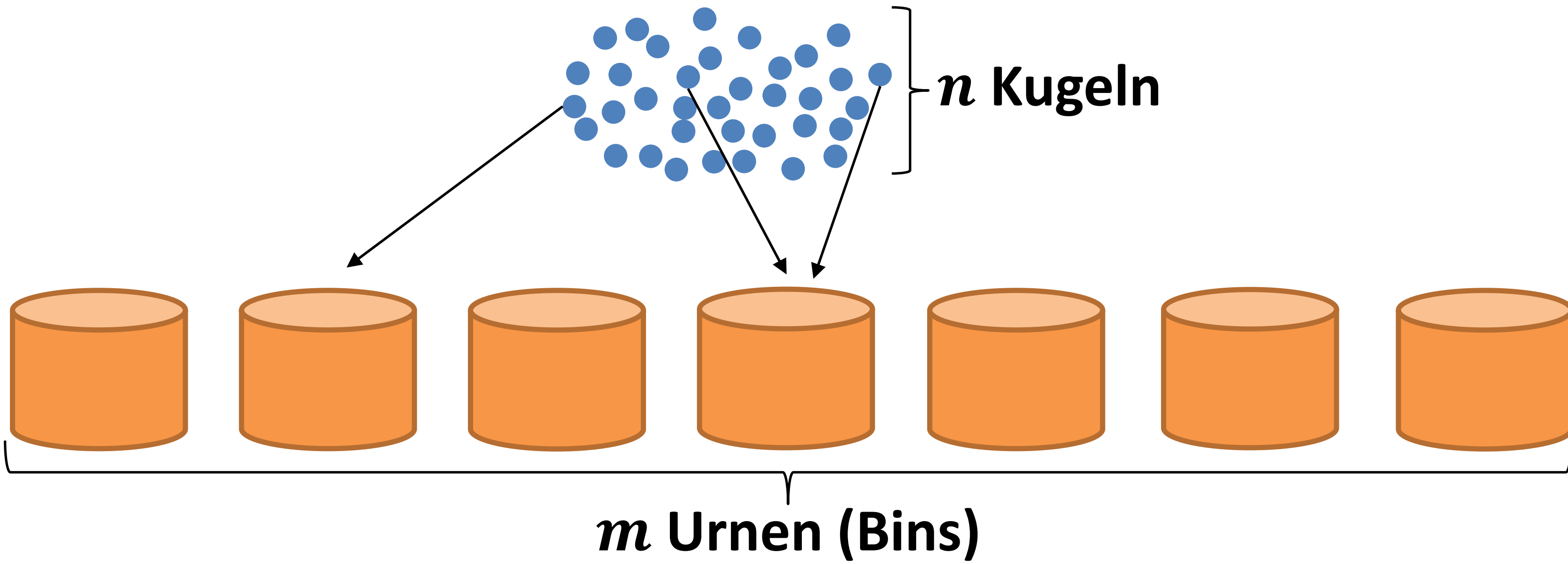

$\alpha = \frac{n}{m}$

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

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

  mit hoher Wahrscheinlichkeit (whp) $\in O\left({}^{n}/_{m} + {}^{\log n}/_{\log\log n}\right)$
  - bei $n \leq m$ also $O\left({}^{\log n}/_{\log\log n}\right)$
- Die längste Liste wird also Länge $\Theta\left({}^{\log n}/_{\log\log n}\right)$ haben.

# Balls and Bins




**Erwartete Laufzeit (für jeden Schlüssel):**

- Schlüssel in Tabelle:
  - Liste eines zufälligen Eintrags
  - entpricht 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:

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

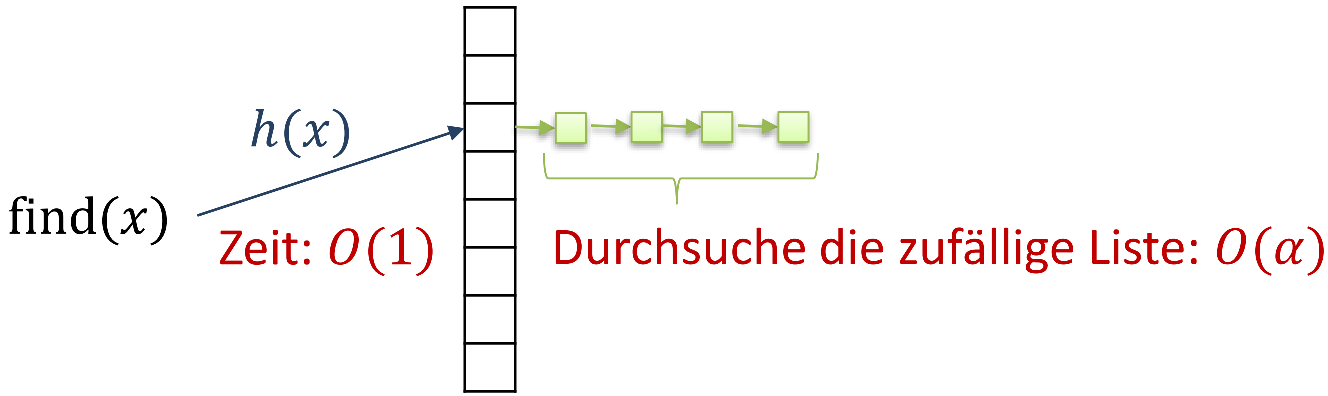
## Kosten einer Suche:

- 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: $\boldsymbol{O(1+\alpha)}$



# Erwartete Laufzeit von Find

**Load $\alpha$ der Hashtabelle:**

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

**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: $\boldsymbol{O(1 + \alpha)}$**

# Laufzeiten Hashing mit Chaining

**create:**

- Laufzeit $O(1)$

**insert, find & delete:**

- Worst Case: $\boldsymbol{\Theta(n)}$
- Worst Case mit hoher Wahrsch. (bei zufälligem $h$): $\boldsymbol{O\left(\alpha + \frac{\log n}{\log\log n}\right)}$

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

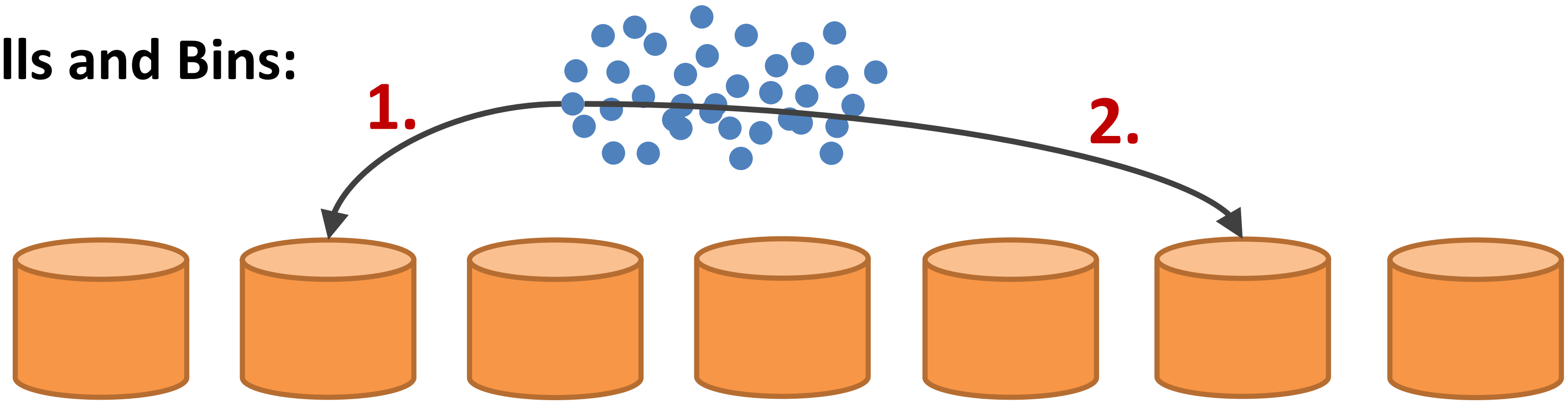
- Erwartete Laufzeit (für bestimmten Schlüssel $x$): $\boldsymbol{O(1+\alpha)}$
  - gilt für erfolgreiche und nicht erfolgreiche Suchen
  - Falls $\alpha = O(1)$ (d.h., Hashtabelle hat Grösse $\Omega(n)$), dann ist das $O(1)$
- Hashtabellen sind extrem effizient und haben **typischerweise $O(1)$ Laufzeit 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"

# 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

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

  - 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

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

# Lineares Sondieren

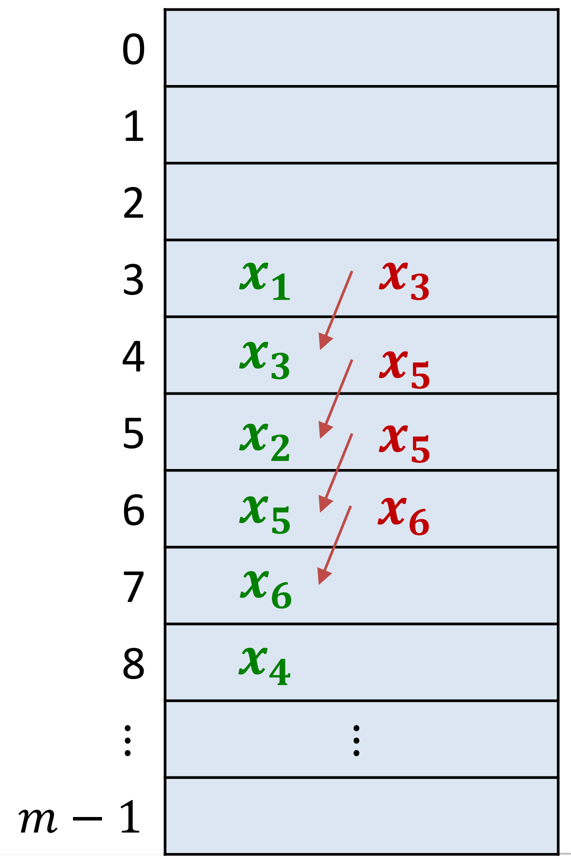
**Idee:**

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

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

für $i = 0, \dots, m-1$

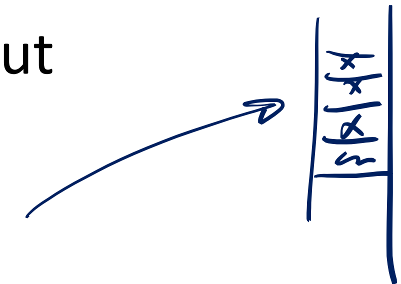
- **Beispiel:**
  Füge folgende Schlüssel ein
  - $x_1, h(x_1) = 3$
  - $x_2, h(x_2) = 5$
  - $x_3, h(x_3) = 3$
  - $x_4, h(x_4) = 8$
  - $x_5, h(x_5) = 4$
  - $x_6, h(x_6) = 6$
  - …



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

# Quadratisches Sondieren

**Idee:**

- Nehme Sequenz, welche nicht zu Cluster führt:

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

für $i = 0, \dots, m-1$

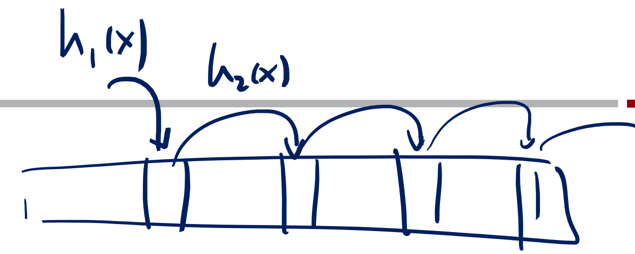
**Vorteil:**

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

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

- 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

# Doppel-Hashing




**Idee:** Benutze zwei Hashfunktionen

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

**Vorteile:**

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

**Offene Adressierung:**

- Schlüssel $x$ kann an folgenden Positionen sein:

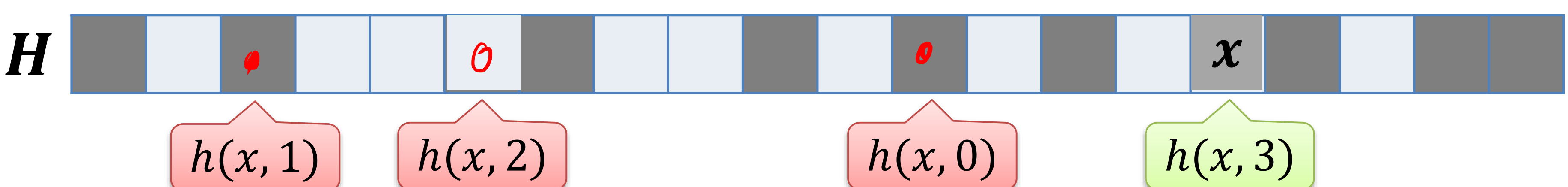
$$h(x,0), h(x,1), h(x,2), \ldots, h(x,m-1)$$

**Operation Find?**

Hashtabelle

```
i = 0
while i < m and H[h(x,i)] != None and H[h(x,i)].key != x:
  i += 1
return i < m and H[h(x,i)] != None and H[h(x,i)].key == x
```

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 Delete

**Offene Adressierung:**

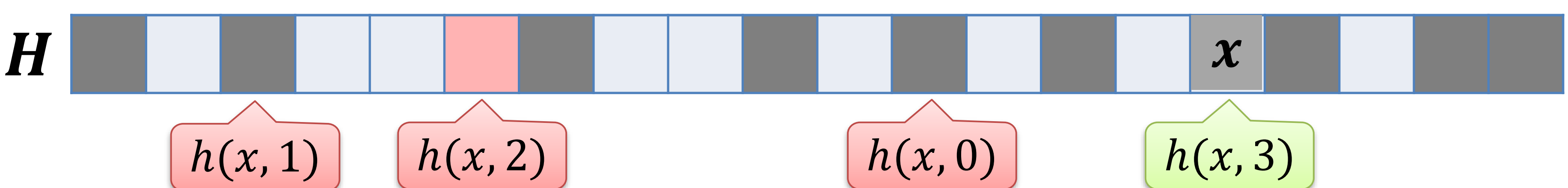
- Schlüssel $x$ kann an folgenden Positionen sein:

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

**Operation Delete**

```
i = 0
while i < m and H[h(x,i)] != None and H[h(x,i)].key != x:
  i += 1
if i < m and H[h(x,i)] != None and H[h(x,i)].key == x:
  H[h(x,i)] = 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:**

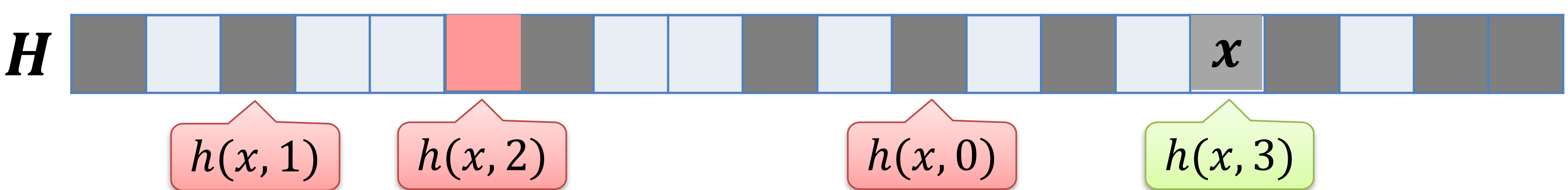
- Schlüssel $x$ kann an folgenden Positionen sein:

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

**Operation Find**

```
i = 0
while i < m and H[h(x,i)] != None and H[h(x,i)].key != x:
  i += 1
return i < m and H[h(x,i)] != None and H[h(x,i)].key == x
```

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 : Zusammenfassung

**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

# Zusammenfassung Hashing

**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.

# Hashing in Python

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*)

# Hashing in Java

**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 Schlüsseln

# Hashing in C++

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

**unordered_map:**

- Seit C++11 in Standard STL

http://www.cplusplus.com/reference/unordered_map/unordered_map/

# Hashing in C++

**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*)

# Hashing in C++

## 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

# Balls Into Bins

- m Urnen, n Kugeln


$B \sim \mathrm{Bin}(n, 1/m)$

$$P(B \geq k) \leq \binom{n}{k} \frac{1}{m^k}$$

$$\leq \left(\frac{en}{mk}\right)^k = e^{-k \ln\left(\frac{mk}{en}\right)}$$

$$\boxed{\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \left(\frac{en}{k}\right)^k}$$

$$\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k\cdot(k-1)\cdots 1}$$

$$\frac{n-i}{k-i} \geq \frac{n}{k}$$

$$\mathbb{E}[\text{Urnen mit} \geq k \text{ Kugeln}] \leq n \cdot e^{-k \ln\left(\frac{mk}{en}\right)} \overset{!}{<} 1$$

$$\Updownarrow$$

$$k \cdot \ln\left(\frac{k}{e\alpha}\right) = k \cdot \ln\left(\frac{mk}{en}\right) > \ln n$$

$$\alpha = O(1) \rightarrow k \cdot \ln\left(\frac{k}{c}\right) > \ln n \rightarrow k = \Theta\left(\frac{\log n}{\log\log n}\right)$$

$$\alpha = \log^{1-\varepsilon}(n) \qquad k \cdot \ln\left(\frac{k}{e \cdot \log^{1-\varepsilon} n}\right) > \ln n$$

$$k \ln\left(\frac{k}{e\alpha}\right) > \ln n \qquad \alpha = \lambda \cdot \log n \qquad \lambda \in [0,1]$$

$$\boxed{k = c \cdot \frac{\ln n}{\ln(1/\lambda)}}$$

$$c \cdot \frac{\log n}{\log(1/\lambda)} \cdot \left[\ln c + \cancel{\ln\ln(n)} - \ln\ln(1/\lambda) - 1 + \ln(1/\lambda) - \cancel{\ln\ln(n)}\right] \overset{!}{>} \ln n$$

$$\alpha > \log n \longrightarrow \text{max. Urne} = \Theta(\alpha)$$