Informatik II - SS 2018

(Algorithmen & Datenstrukturen)

Vorlesung 6 (7.5.2018)

Dictionaries, Binäre Suche, Hashtabellen I

UNI FREIBURG

Fabian Kuhn / Yannic Maus

Algorithmen und Komplexität

Abstrakte Datentypen : Dictionary

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*

Dictionary

UNI FREIBURG

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

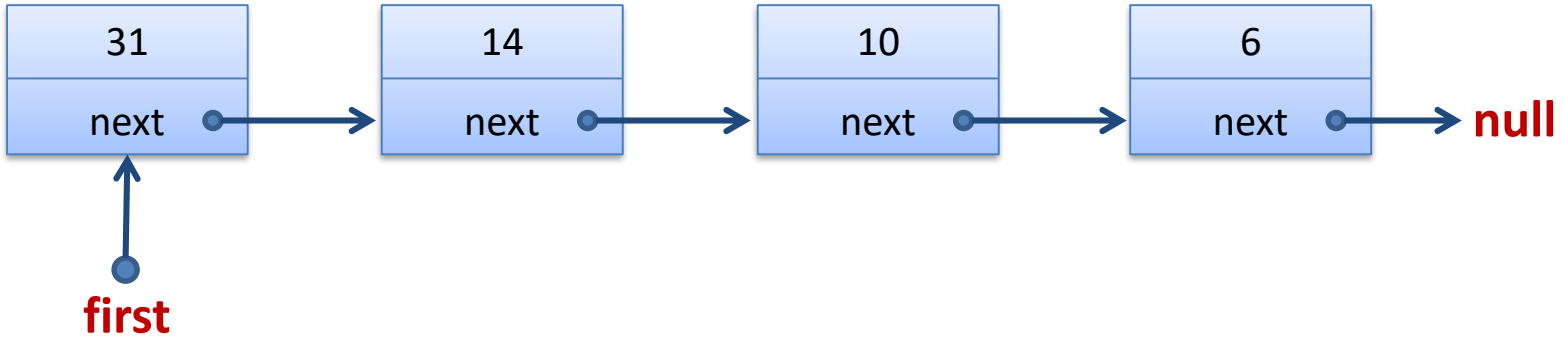
Dictionary Beispiele:

- 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)

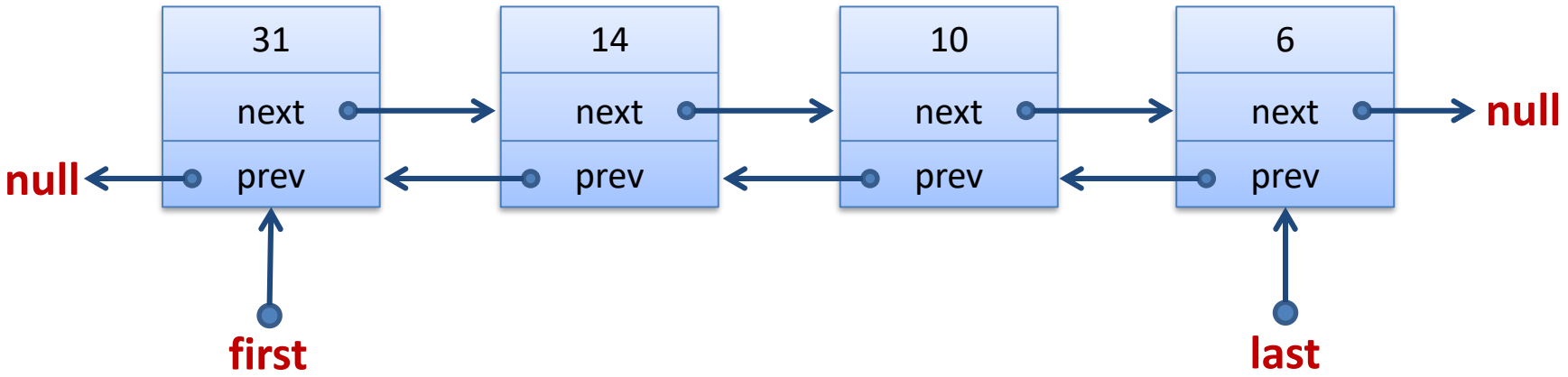
In all diesen Fällen ist insbesondere eine schnelle *find*-Op. wichtig!

Verkettete Listen: Struktur

Einfach verkettete Liste (Singly Linked List):



Doppelt verkettete Liste (Doubly Linked List):



Dictionary mit verketteten Listen

UNI FREIBURG

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 verketten Listen muss man stoppen, sobald *current.next.key == key* ist!

Dictionary mit verketteten Listen

UNI FREIBURG

Laufzeiten:

create: $O(1)$

insert: $O(1)$

find: $O(n)$

delete: $O(n)$

Ist das gut?

Dictionary mit Array

UNI FREIBURG

Operationen:

- *create*:
  - lege neues Array der Länge *NMAX* an

O(1)

- *D.insert(key, value)*:
  - füge neues Element hinten an (falls es noch Platz hat)
  - Annahme: Es gibt noch keinen Eintrag mit dem Schlüssel *key*

O(1)

- *D.find(key)*:
  - gehe von vorne (oder hinten) durch die Elemente

O(n)

- *D.delete(key)*:
  - suche zuerst nach dem *key*
  - lösche Element dann aus dem Array:

O(n)

Man muss alles dahinter um eins nach vorne schieben!

Dictionary mit Array

Laufzeiten:

create: $O(1)$

insert: $O(1)$

find: $O(n)$

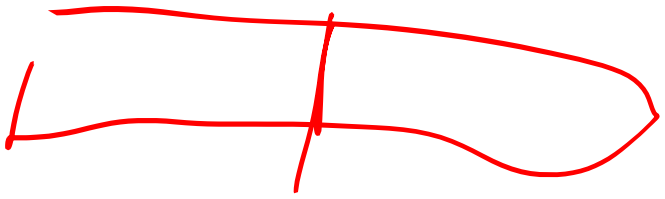
delete: $O(n)$

Bessere Ideen?

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:



Mitte, vergleicht man das zu suchende Element mit dem Element in d. Mitte → suche „links" oder „rechts" weiter

Binäre Suche

Benutze Divide and Conquer Idee!

Suche nach der Zahl (dem Key) 19:

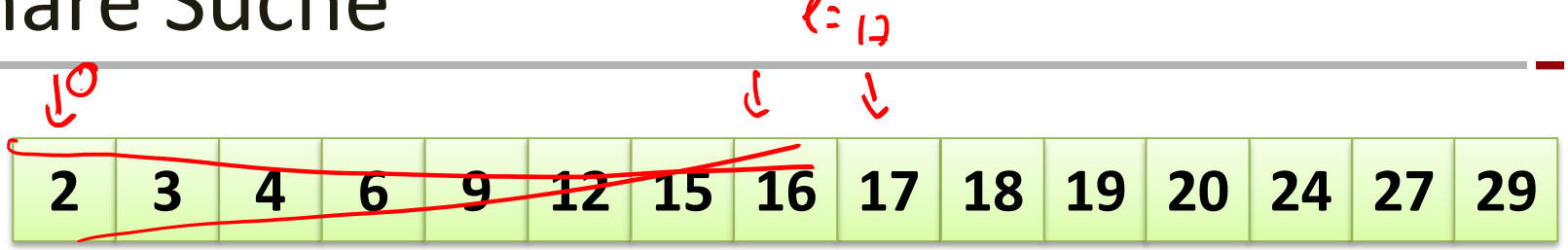
2	3	4	6	9	12	15	16	17	18	19	20	24	27	29

$16 < 19$ $\quad 20 < 19$

$18 < 19$

$O(\log n)$

Binäre Suche



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 = r = m
```

Binäre Suche

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 $A[l] = x$

Ist der Algorithmus korrekt?

UNI FREIBURG

Wie überprüft man das?

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

Hoare Kalkül

- **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?

UNI FREIBURG

Vorbedingung

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

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

Schleifeninvariante

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

Nachbedingung

- *Falls x im Array ist, dann gilt $A[l] = x$*
- Abbruchbedingung while-Schleife → $l \geq r$ 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

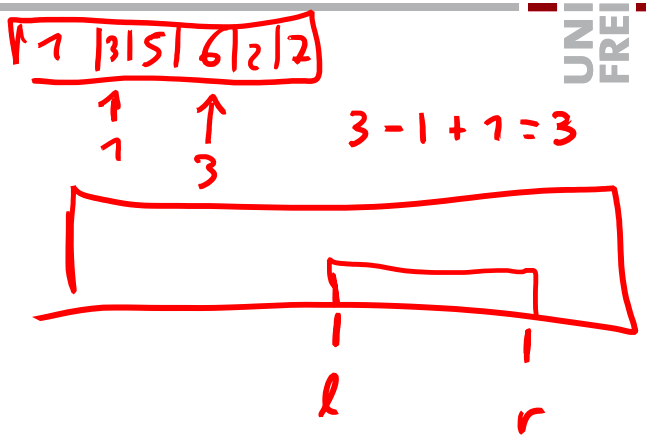
- *Falls x im Array ist, dann gilt $A[l] \leq x \leq A[r]$*

$l = m+1$: ja, suchen im rechten Bereich weiter, da $A[m] < x$

$r =$ analog.

Terminiert der Algorithmus?

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



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

1. – $l = m + 1$:

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

2. – $r = m - 1$:

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

3. – Sonst wird x gefunden und $r - l + 1$ wird 1

Laufzeit

Terminiert der Algorithmus?

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

Laufzeit?

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

$T(n) \leq T(\frac{n}{2}) + c \leq T(\frac{n}{4}) + c + c = T(\frac{n}{4}) + 2c$

$\leq T(\frac{n}{8}) + 3c \quad \ldots \leq T(\frac{n}{2^\ell}) + \ell \cdot c$

$(\ell = \log n)$

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

$T(1) \leq c$

Laufzeit Binäre Suche

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

$T(n) \leq T\left(\frac{n}{2}\right) + c \qquad T(1) \leq c$

Per Induktion: IH: $T(n) \leq (1 + \log n) \cdot c$

Induktionsanfang: $T(1) \leq c = (1 + \log 1) \cdot c$

Schritt:

$$T(n) \leq T\left(\frac{n}{2}\right) + c \overset{IH}{\leq} \left(1 + \log\left(\frac{n}{2}\right)\right) \cdot c + c$$

$\log\left(\frac{n}{2}\right) = \log(n) - \log(2) = \log n - 1$

$$= \left(2 + \log\left(\frac{n}{2}\right)\right) \cdot c = (1 + \log n) \cdot c$$

Finden ein Element in Zeit

$T(n) = (1 + \log(n)) \cdot c = O(\log n)$

Dictionary mit sortiertem Array

UNI FREIBURG

Operationen:

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

Dictionary mit sortiertem Array

Laufzeiten:

create:

insert:

find:

delete:

Können wir alle Operationen schnell machen?

- und das *find* noch schneller?

Dictionary mit sortiertem Array

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?

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	**Yannic**
7	**Value 3**
8	None
⋮	⋮
$M-1$	None

find(2) → *"Value 1"*

insert(6, "Yannic")

delete(4)

Direkte Adressierung : Probleme

UNI FREIBURG

1. Direkte Adressierung benötigt zu viel Platz!

- Falls Schlüssel ein beliebiger *int* (32 bit) sein kann:

  Wir benötigen ein Array der Grösse $2^{32} \approx 4 \cdot 10^9$. $\approx 0{,}5$ GB

  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"

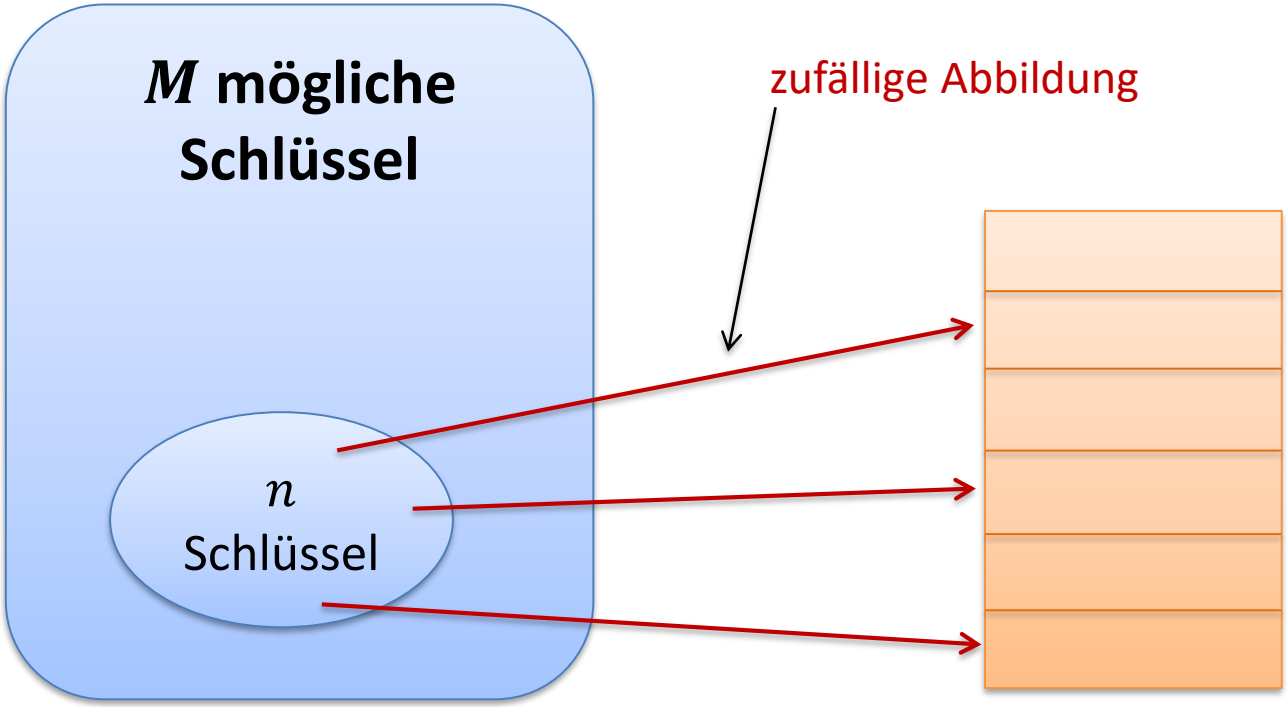
  "Alles" kann als Folge von Bits abgespeichert werden:
  Interpretiere Bit-Folge als ganze Zahl

- **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

UNI FREIBURG

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

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

$S >>>> m$

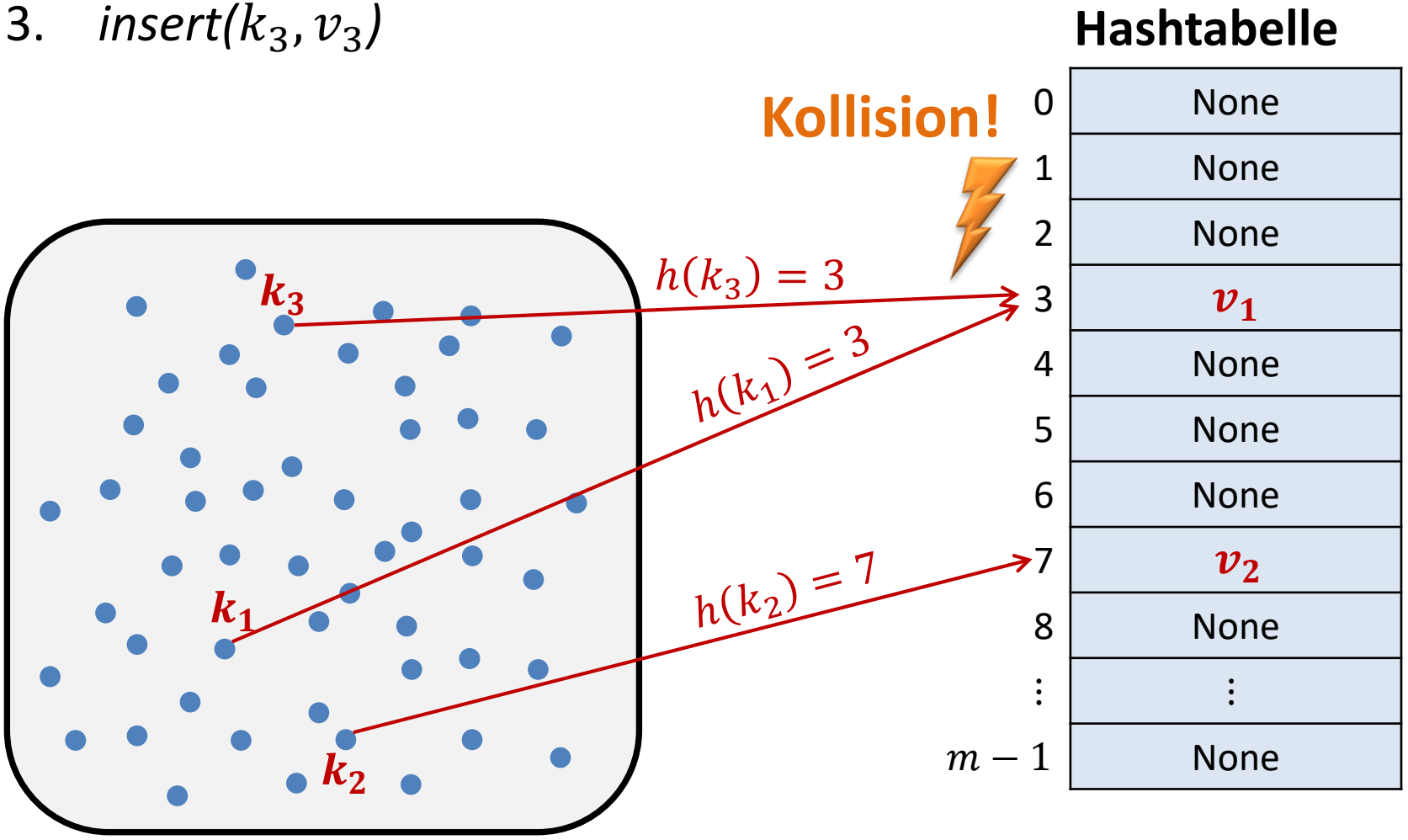
Hashfunktion

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

- 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 : Kollisionen

UNI FREIBURG

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?

- Eine andere Hashfunktion nehmen?

- Weitere Ideen?

Hashtabellen : Kollisionen

UNI FREIBURG

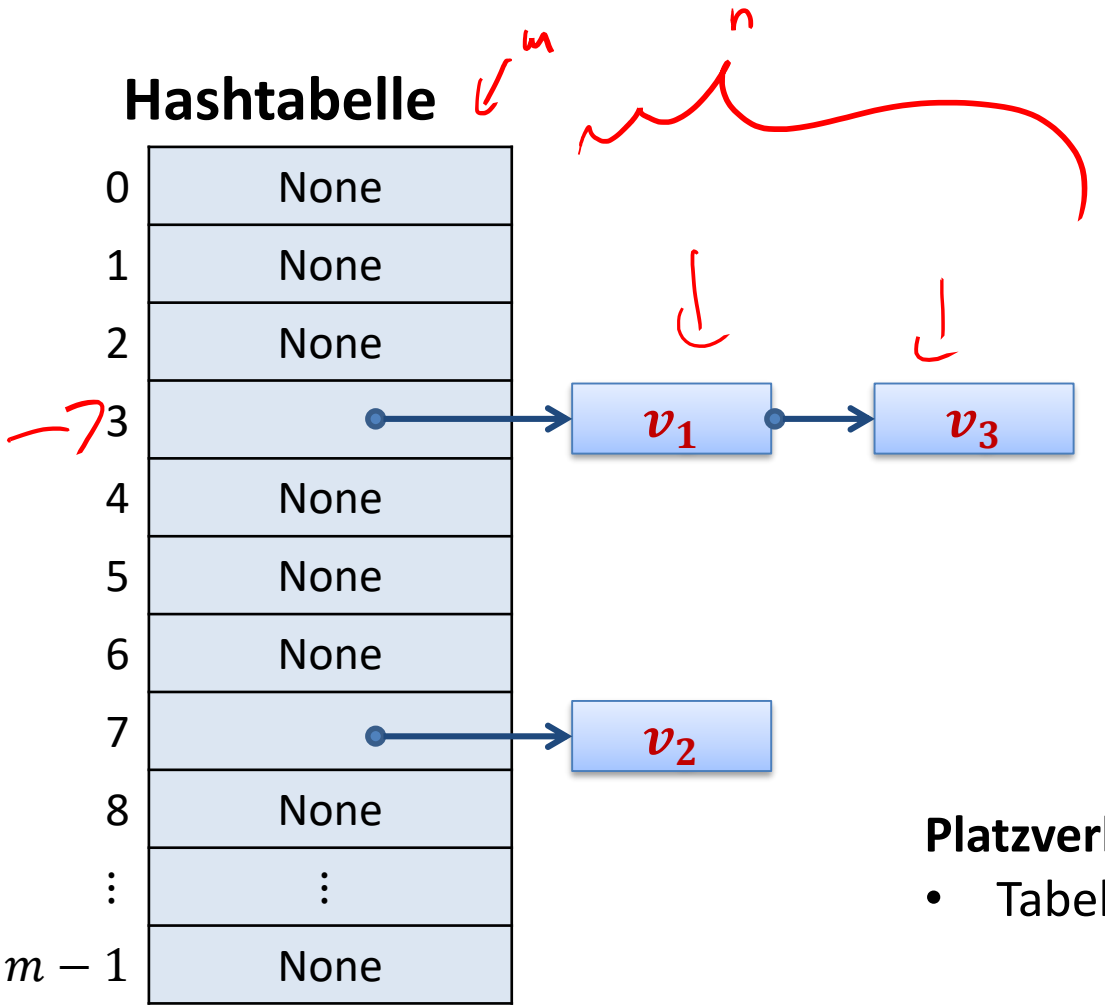
Kollisionen Lösungsansätze

- 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



Platzverbrauch:

- Tabellengrösse m, Anz. Elemente n

$O(m+n)$

Laufzeit Hashtabellen-Operationen

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: $O(1)$ (vorne in der Liste einfügen)

find: worst case $O(n)$

find(x)

$1 +$ Länge der Liste an $\overbrace{Pos(x)}^{h(x)}$

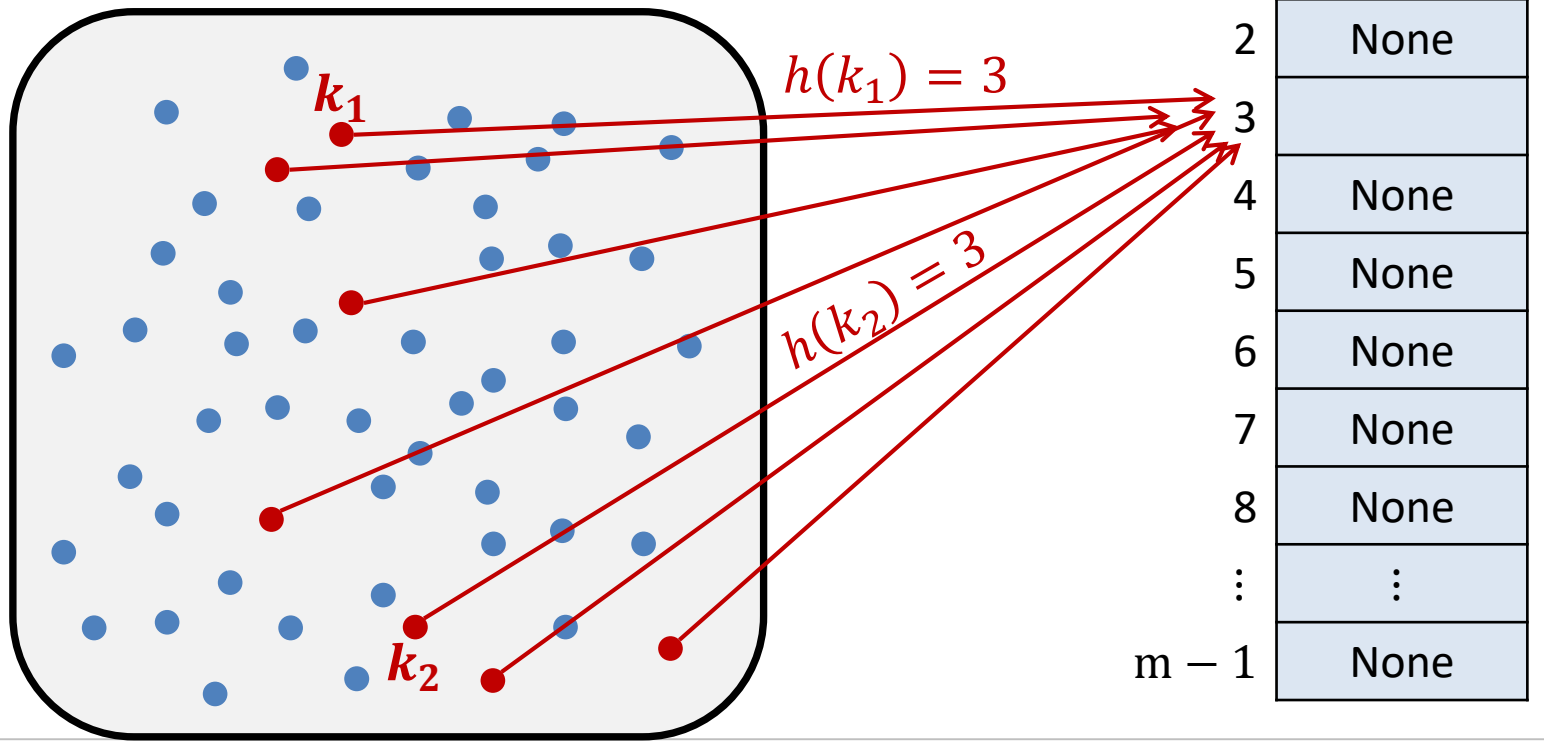
Im Durchschnitt $\frac{n}{m}$

delete:

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



Länge der verketten Liste

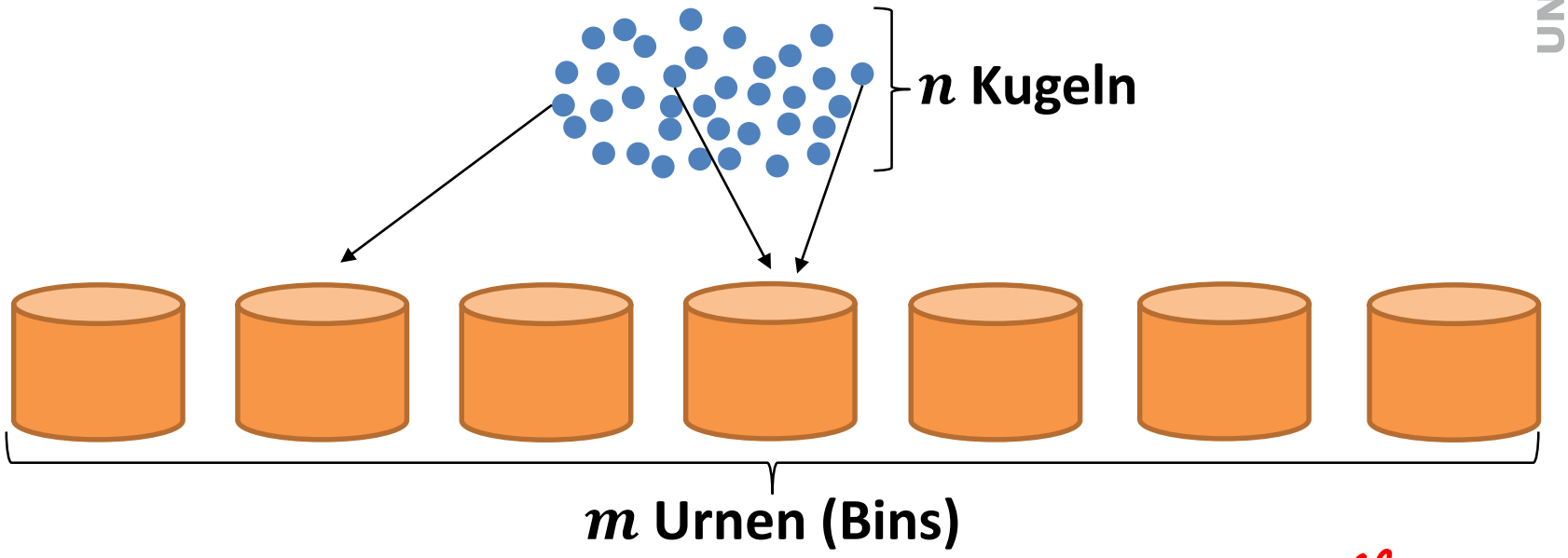
- Kosten von *find* und *delete* hängt von der Länge der entprechenden Liste ab
- Wie lang werden die Listen
  - Annahme: Grösse der Hashtabelle m, Anzahl Elemente n
  - Weitere Annahme: Hashfunktion h verhält sich wie zufällige Funktion
- Listenlängen entspricht folgendem Zufallsexperiment

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):
  - Schlüssel in Tabelle: entpricht der #Kugeln in der Urne einer zufälligen Kugel
  - Schlüssel nicht in Tabelle: #Kugeln einer zufälligen Urne

Erwartete Laufzeit von Find

Load α der Hashtabelle:

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

n: # Anzahl Schlüssel, die ich speichern möchte

m: ← Größe d. Hashtabelle

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:

Erwartete Laufzeit von Find

UNI FREIBURG

Load α 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:

Laufzeiten Hashing mit Chaining

Zusammenfassung Laufzeiten:

create & insert:

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

find & delete:

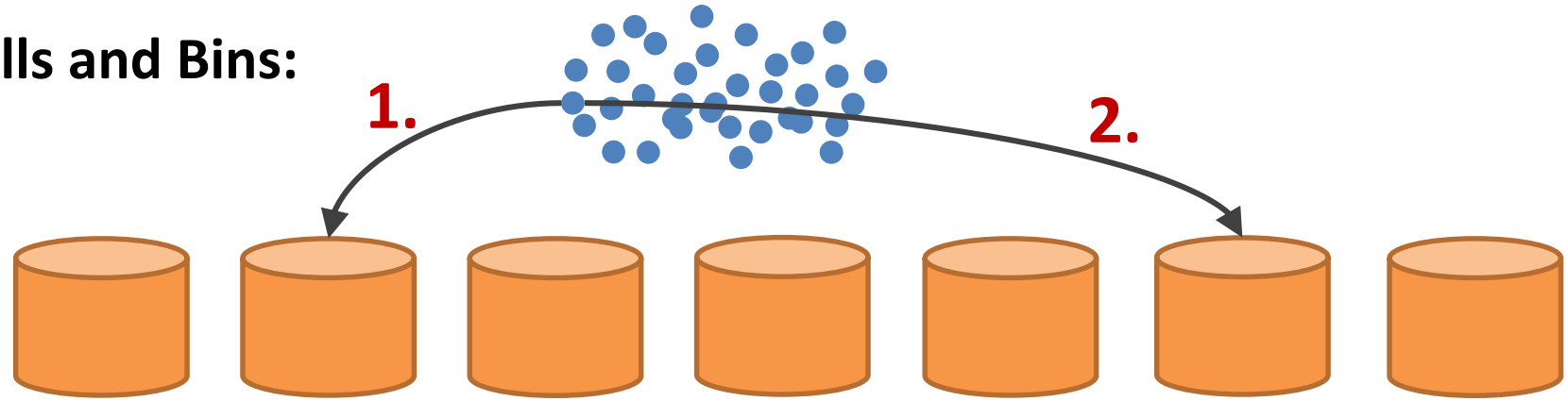
- Worst Case: $\boldsymbol{\Theta(n)}$
- Worst Case mit hoher Wahrsch. (bei zufälligem h): $\boldsymbol{O\left(\alpha + \frac{\log n}{\log \log n}\right)}$
- Erwartete Laufzeit (für bestimmten Schlüssel x): $\boldsymbol{O(1+\alpha)}$
  - gilt für erfolgreiche und nicht erfolgreiche Suchen
  - Falls $\alpha = O(1)$ (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"