



Algorithmen und Datenstrukturen

# Kapitel 08: Hashing

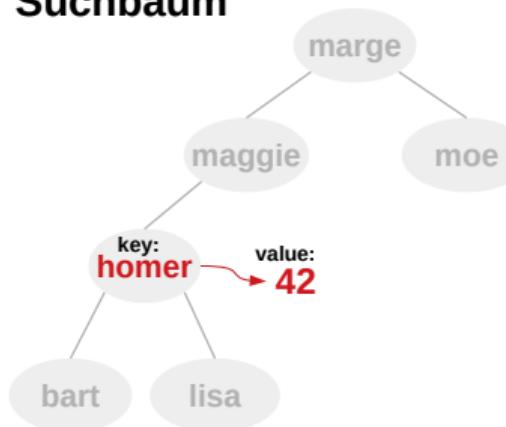
Prof. Dr. Adrian Ulges

B.Sc. \*Informatik\*  
Fachbereich DCSM  
Hochschule RheinMain

# Mengen und Dictionaries

Beispiel: `map.get('homer') = 42`

## Suchbaum



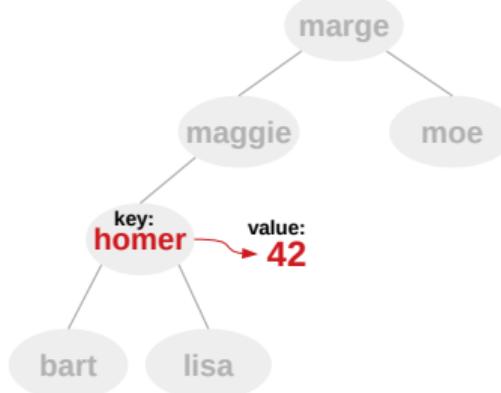
## Hashing

key	value
marge	39
bart	42
moe	45
lisa	8
homer	42
maggie	1

- ▶ **Ziel:** Realisiere **Mengen** (sets) und **Dictionaries** (maps).
- ▶ **Bisher:** Suchbäume (siehe JCF: TreeSet, TreeMap).
- ▶ **Im Folgenden:** Eine weitere Standard-Strategie, **Hashing**.
- ▶ **Im JCF:** HashSet, HashMap.

# Hashing

## Suchbaum



## Hashing

key	value
marge	39
bart	42
moe	45
lisa	8
homer	42
maggie	1

## Suchbäume

- ▶ basieren auf **Vergleichen**
- ▶ **dynamische** Datenstruktur
- ▶ **logarithmische** Operationen  
(bei **ausgeglichenen** Bäumen)

## Hashing

- ▶ basiert auf **Hash-Funktion**
- ▶ **statische** Datenstruktur (= *Array*)
- ▶ **meist konstante** Operationen  
(können entarten!)

# Hashing: Grundprinzip



- ▶ Beim Hashing bestimmt der **Schlüssel** (key) die **Position** innerhalb eines Arrays.
  - ▶ Das Array – die sogenannte **Hash-Tabelle** – hat  $N$  Plätze ( $0, 1, 2, \dots, N - 1$ ).

## Beispiel: int-Schlüssel

- **Beispiel:** Matrikelnummern (z.B. 343219)
  - $N$  = Größe des Schlüssel-**Wertebereichs**
  - **Hier:**  $N = 2^{32}$  (ca. 4.3 Mrd).

## Vorteile / Nachteile?

- ☺ Zugriff ist **günstig** ( $O(1)$ , Position im Speicher einfach berechenbar).
  - ☺ keine Vergleiche notwendig.
  - ☹☹ **hoher Speicherbedarf**.

# Hashing: Grundprinzip

Wie realisieren wir eine **kleinere Hash-Tabelle**?

Beispiel: **int-Schlüssel**

- Wir wählen ein **kleineres N**.
- Wir bestimmen die **Position** eines **Schlüssels k** im Array als

key	value
0	
1	
...	
219	<b>343219    obj</b>
999	

$$h(k) = k \% N$$

Beispiel (links)

- Die Tabelle besitze  $N = 1000$  Einträge.
- Schlüssel  $k = \underline{343219}$ .
- $h(k) = k \% 1000 = 219$ .



# Outline

1. Hash-Funktionen

2. Kollisionsbehandlung 1: Sondieren

3. Kollisionsbehandlung 2: Verkettung

# Hashing: Grundprinzip

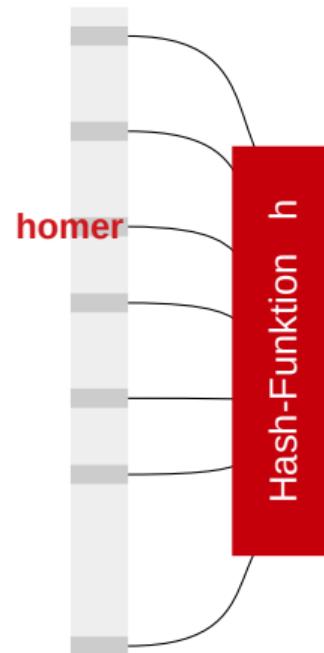
Schlüssel = beliebige Objekte!

- ▶ Im allgemeinen Fall sind Schlüssel keine int-Werte, sondern **beliebige Objekte**.
- ▶ Auch diese müssen wir auf Positionen in der Hash-Tabelle abbilden.
- ▶ **Achtung:** Der Schlüssel-Wertebereich kann **unendlich groß** sein!
- ▶ **Beispiel:** Die Menge aller Strings.

Ansatz: Schlüsseltransformation

- ▶ Eine **Hash-Funktion h** berechnet aus einem Schlüssel  $k$  eine **Position** in der Hash-Tabelle:  $h(k) \in \{0, 1, \dots, N - 1\}$ .

Wertebereich  
Schlüssel K



Hash-Tabelle T

$p$	key	value
0		
1		
2	homer	42
3		
4		
5		
6		

Tabellengröße  
 $N=7$

# Definition: Hash-Funktion

## Definition (Hash-Funktion)

Es sei  $K$  eine Menge (bzw. ein Typ) von Schlüsseln und  $N \in \mathbb{N}^+$  die Größe einer Hash-Tabelle. Dann ordnet eine **Hash-Funktion**

$$h: K \rightarrow \{0, 1, \dots, N - 1\}$$

einem Schlüssel  $k$  einen Wert  $h(k)$  zu.

## Anmerkungen

- ▶ Der **Schlüssel-Typ  $K$**  kann beliebig sein: Strings, Zahlen, Objekte beliebiger Klassen ...
- ▶ Weil die Hash-Funktion bei jedem Einfügen/Suchen/Löschen verwendet wird, sollte sie **schnell berechenbar** sein ( $O(1)$ ).
- ▶ Das Ergebnis  $h(k)$  bezeichnen wir auch als **Hash-Wert** von  $k$ .

# Hashing: Naive Implementierung

```
class HashMapNaive<K,V> {  
  
    private class Entry {  
        K key;  
        V value;  
    }  
  
    Entry[] table;  
  
    public HashMapNaive(int N) {  
        table = (Entry[]) new Object[N];  
    }  
  
    private int hashCode(K key) {  
        ...  
    }  
  
    public V get(K key) {  
        return table[hashCode(key)].value;  
    }  
  
    public void insert(K key, V value) {  
        table[hashCode(key)] = new Entry(key,  
                                         value);  
    }  
  
    public void delete(K key) {  
        table[hashCode(key)] = null;  
    }  
}
```

- ▶ Typen K (key) und V (value) für Schlüssel und Wert.
- ▶ Ein Entry steht für ein **Schlüssel-Wert-Paar**.
- ▶ Die Hash-Tabelle ist ein **Array** solcher Schlüssel-Wert-Paare.
- ▶ **Einfügen, Löschen, Suchen** sind ähnlich:
  - ▶ Hash-Code berechnen
  - ▶ Feld in Tabelle bearbeiten.

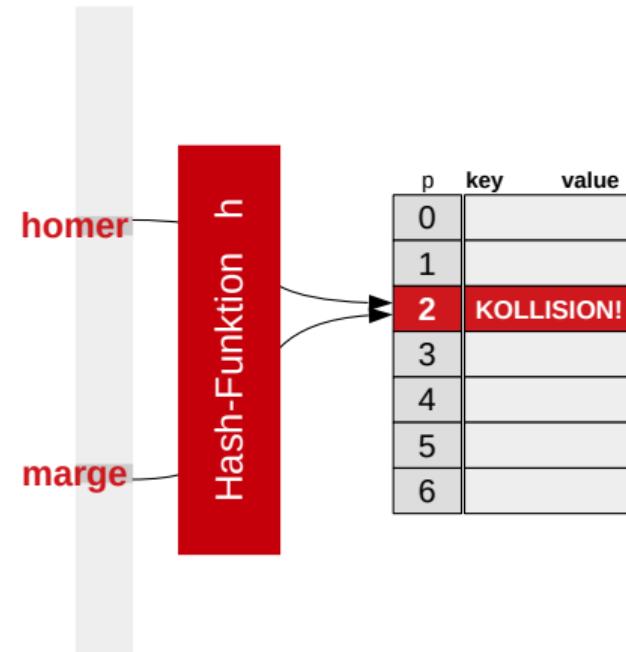
# Problem: Kollisionen

- ▶ Meist gilt  $N \ll \#K$   
*(die Tabelle ist deutlich kleiner als die Menge möglicher Schlüssel).*
- ▶ Gemäß dem **Taubenschlagprinzip** folgt:

Mindestens **zwei verschiedene Schlüssel** müssen auf **identische Hash-Werte** (Positionen) abgebildet werden.

- ▶ Hash-Funktionen sind also **nicht injektiv!**
- ▶ Wir bezeichnen solche Konflikte als **Kollisionen**.

Wertebereich  
Schlüssel K

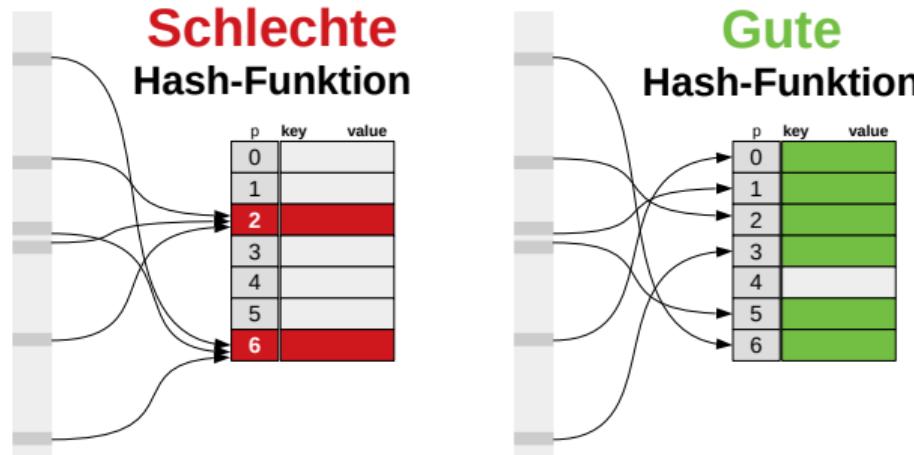


# Strategien gegen Kollisionen

1. **Reduziere** die Zahl der Kollisionen mit einer “guten” Hash-Funktion.
2. **Behandle** die verbleibenden Kollisionen. **Zwei Strategien** (*später*): Verkettung und Sondieren.

Strategie 1: “Gute” Hash-Funktionen...

... besitzen eine **hohe Streuung**: Schlüssel werden **gleichmäßig** auf Hash-Werte verteilt, seltene Kollisionen.



# Hash-Funktionen für Strings: Beispiele

Wir betrachten ein paar **Beispiel-Hash-Funktionen**:

## 1. int-Schlüssel

$$h(k) = \lfloor k/N \rfloor \% N$$

$\downarrow$   
(abgerundet)

Bsp.:  $N=100$

$$\begin{aligned} h(\cancel{1743}) &= 17 \\ h(\cancel{1786}) &= 17 \end{aligned}$$

↳ Kollision

## 2. int-Schlüssel, Version 2

$$h(k) = k \% N$$

Bsp.:  $N=5$

$k$	1	2	3	4	5	6	7	8	...
$h(k)$	1	2	3	4	0	1	2	3	...

$$N=1024 = 2^{10}$$

$$h(0101\cancel{1000100010}) = 1000100010$$

:(  $h(1010\cancel{1000100010}) =$  ??

# Hash-Funktionen

Die Wahl von **N** ist entscheidend für das Streuverhalten!

Schlechte Werte für *N* ...

- ▶ ... sind z.B. **Zweierpotenzen** (*siehe oben*).
- ▶ Für  $N = 2^i$  werden nur die letzten  $i$  Bits des Hash-Codes verwendet, die restlichen Bits werden ignoriert.

Gute Werte für *N* ...

- ▶ ... sind z.B. **Primzahlen**.
- ▶ Auch bei gehäuften Schlüsseln / Teilbitfolgen gute Streuung.

# Hash-Funktionen für Strings: Beispiele

## 3. String-Schlüssel

- Wir hashen die Studis der HSRM, mit dem **Nachnamen** als Schlüssel.
- Es sei  $k = k_1, k_2, \dots, k_m$  ein Schlüssel.
- Wir interpretieren die **Buchstaben**  $k_i$  als **Zahlen** (ASCII-Codes, Unicodes).
- Wir nutzen die **ersten drei** Buchstaben:  
$$h(k) = (k_1 + k_2 + k_3) \% N$$

$$h("ULGES") = (U + L + G) \% N$$

as    4C    47

$$h(\underline{\text{SCHUSTER}}) = h(\underline{\text{SCHNEIDER}})$$

↳

Character	Hex	Character	Hex
A	41	N	4E
B	42	O	4F
C	43	P	50
D	44	Q	51
E	45	R	52
F	46	S	53
G	47	T	54
H	48	U	55
I	49	V	56
J	4A	W	57
K	4B	X	58
L	4C	Y	59
M	4D	Z	5A

## Hash-Funktionen für Strings: Beispiele

Character	Hex	Character	Hex
A	41	N	4E
B	42	O	4F
C	43	P	50
D	44	Q	51
E	45	R	52
F	46	S	53
G	47	T	54
H	48	U	55
I	49	V	56
J	4A	W	57
K	4B	X	58
L	4C	Y	59
M	4D	Z	5A





# Hash-Funktionen für Strings

**Nicht-numerische** Attribute müssen wir für die Hash-Funktion auf Zahlen **abbilden**. Beispielhaft tun wir dies für **Strings**:

## Definition (Hash-Funktion für Strings (Version 1))

Gegeben sei ein String  $k = k_1, k_2, \dots, k_n$ . Wir interpretieren die Buchstaben  $k_i$  (bzw. ihren ASCII-Code/Unicode) als **Ziffern**. Als Hash-Wert ergibt sich:

$$h(k) = (k_1 \cdot B^{n-1} + k_2 \cdot B^{n-2} + \dots + k_{n-1} \cdot B^1 + k_n \cdot B^0) \% N,$$

gegeben eine **Basis**  $B \in \mathbb{N}^+$  mit  $B > 1$ .

## Beispiel

Basis  $B=10$ , String  $k=“ADS”$ .

$$h(“ADS”) = (A \cdot 100 + D \cdot 10 + S \cdot 1) \% N$$

## Hash-Funktionen für Strings (cont'd)



## Anmerkungen

- Der Hash-Wert hängt (bei gutem  $N$ ) von **allen Buchstaben** des Strings ab. ☺
  - Häufige Wahl für Basis:  $B = 31$  (vgl. Javas `String.hashCode()`).

## Problem

- ▶ Bei langen Strings sind die einzelnen Summanden **sehr groß**. Es kommt zu **Überläufen**.
  - ▶ **Beispiel (37 Buchstaben, Basis 10):**

*h(“With power comes great responsibility”)*

$$= W \cdot 10^{36} + i \cdot 10^{35} + t \cdot 10^{34}$$

## Hash-Funktionen für Strings (cont'd)

Um diese Überläufe zu vermeiden, nutzen wir das **Horner-Schema:**

$$\left( k_1 \cdot B^{n-1} + k_2 \cdot B^{n-2} + \dots + k_{n-1} \cdot B^1 + k_n \cdot B^0 \right) \% N$$

$$\left( (((((k_1 \cdot B) + k_2) \cdot B + k_3) \cdot B + k_4) \cdot B \dots + k_{n-1}) \cdot B + k_n \right) \% N$$

$$\left( (((((k_1 \cdot B + k_2) \% N) \cdot B + k_3 \% N) \cdot B + k_4 \% N) \cdot B + \dots + k_n \% N \right) \% N$$



*Vermeidet  
Überläufe 😊*

Trick: Nach jeder berechneten Stelle führen wir eine %-Operation durch → Es entsteht **kein Überlauf.**

# Hash-Funktionen für Strings (cont'd)

## Definition (Hash-Funktion für Strings (Version 2))

Gegeben sei erneut ein String  $k = k_1, k_2, \dots, k_n$  und eine Basis  $B$ . Wir berechnen denselben Hash-Wert wie in Version 1, vermeiden aber Überläufe mit dem Horner-Schema. Wir rechnen:

```
// Horner-Schema
h ← 0
for i = 1...n:
    h ← ( h * B + ki ) % N // Überlauf vermeiden
return h
```

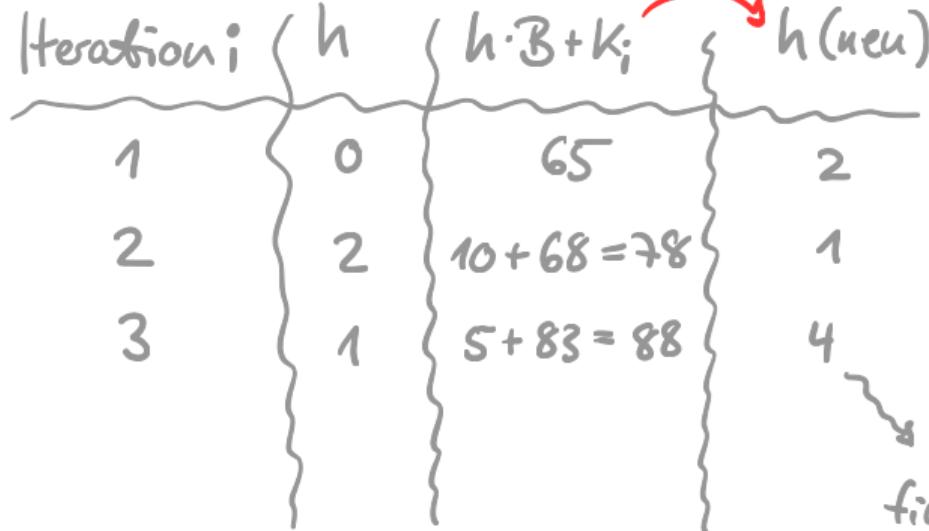
# Beispiel



$B = 5, N = 7, k = \text{"ADS"} (\rightarrow \text{ASCII: } 65, 68, 83)$

$k_1 \ k_2 \ k_3$

$\%N$



"A"  
"D"  
"S"

finales  
Ergebnis

p	key	value
0		
1		
2		
3		
4		"ADS" ...
5		
6		



# Outline

1. Hash-Funktionen
2. Kollisionsbehandlung 1: Sondieren
3. Kollisionsbehandlung 2: Verkettung

# Sondieren

Auch bei guten Hash-Funktionen treten **immer noch Kollisionen** auf (*Taubenschlagprinzip*), und wir müssen sie behandeln.

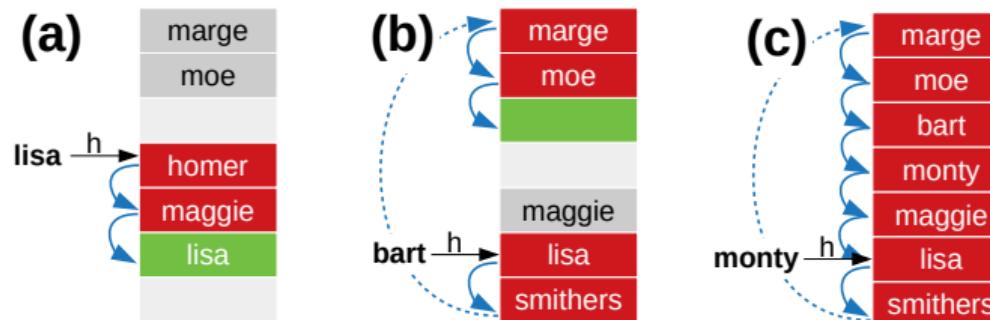
## Lösung 1: Sondieren

Suche (*in festgelegten Schritten*) nach freiem Platz, so lange bis...

- (a) **Schlüssel gefunden**, oder
- (b) **freier Platz** gefunden, oder
- (c) wieder am **Ausgangspunkt** (*Tabelle voll*).

### Einfachste Variante: Lineares Sondieren

Gehe immer **einen Schritt** weiter. Am **Tabellenende**: Umbruch.



# Sondieren: Generell

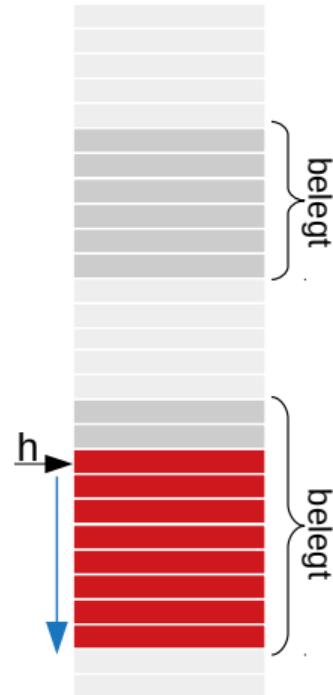
## Problem: Klumpenbildung

- Bei linearer Sondierung **häufen** sich Werte oft in **bestimmten Bereichen** der Tabelle.
- Das widerspricht dem Prinzip der **Gleichverteilung**.
- Such- und Einfüge-Operationen werden **teuer** (*viele Sondierungsschritte!*)!

## Genereller Ansatz: Sondierungsfunktion

Wir definieren eine Funktion  $g : \mathbb{N} \rightarrow \{0, 1, \dots, N - 1\}$ .

- Für jeden Sondierungsschritt  $m$  gibt  $g(m)$  an, **welche Stelle** der Hash-Tabelle geprüft wird.
- $g(0)$  entspricht dem **Hash-Wert**  $h(k)$ .
- Lineares Sondieren:**  $g(m) = (h(k) + m) \% N$ .



# Weitere Sondierungsfunktionen

## Linear, größere Schritte

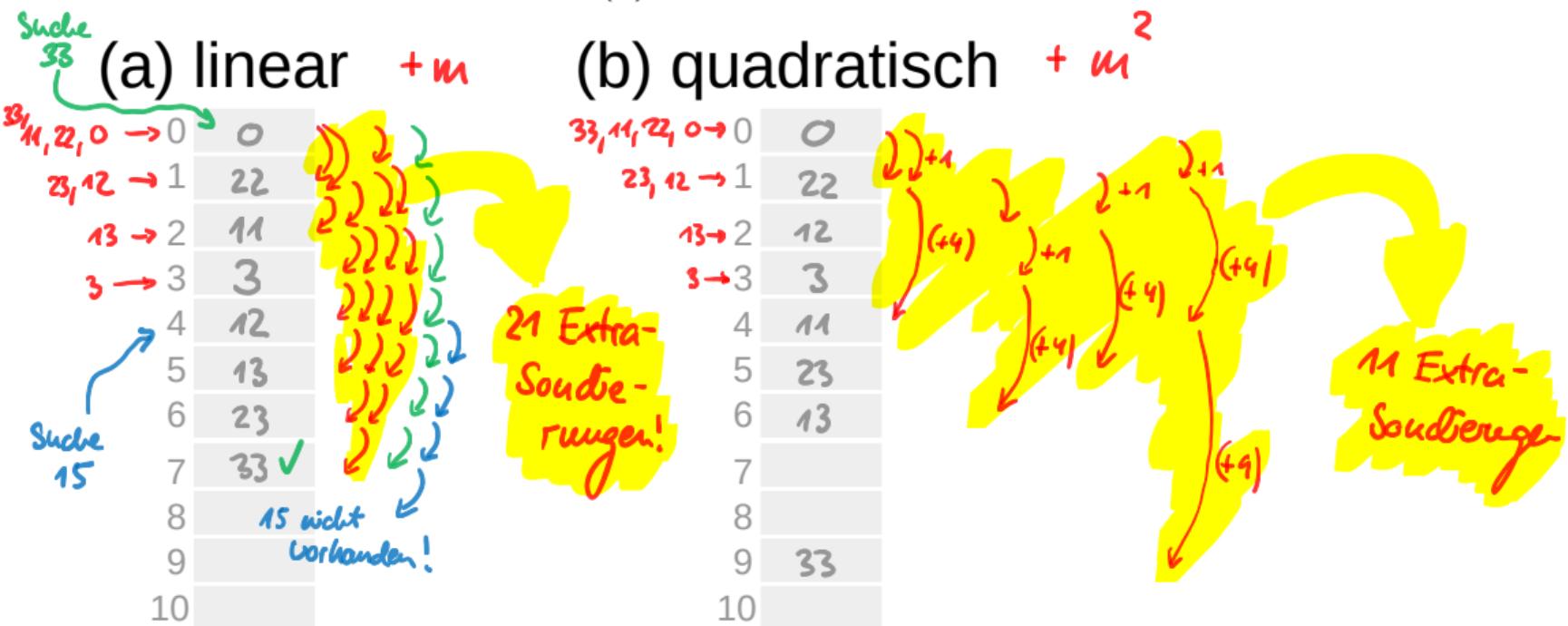
- ▶  $g(m) = (h(k) + c \cdot m) \% N.$
- ▶ Gehe nicht um einen, sondern um  $c$  Schritte weiter.

## Quadratisches Sondieren

- ▶  $g(m) = (h(k) + m^2) \% N.$
- ▶ Quadratische Schritte ( $+1, +4, +9, +16, \dots$ )
- ▶ Sprünge werden mit jedem Schritt größer, weniger “Klumpen”.

# Hashing mit Sondieren: Beispiel

- Tabellengröße  $N = 11$ , Hash-Funktion:  $h(k) = k \% 11$
- Füge ein: 3, 0, 22, 11, 12, 13, 23, 33
- Suche: 33, 15. Lösche: 22 (?)



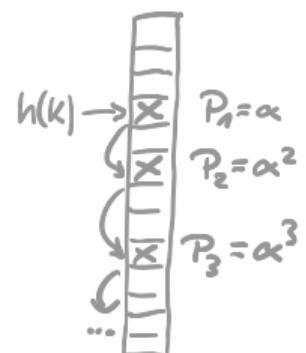
# Sondieren: Aufwandsbetrachtung

## Annahmen

- ▶ **Perfekte Hash-Funktion:** Hash-Werte sind **gleichverteilt** über die Tabelle.
- ▶ Es sei  $M$  die Anzahl der **belegten** Positionen.
- ▶ Wir nennen  $\alpha = M/N$  den **Füllgrad** der Tabelle.

Wieviele Sondierungsschritte brauchen wir im Durchschnitt?

- ▶ Wahrscheinlichkeit dass 1. Position besetzt:  $P_1 = \frac{M}{N} = \alpha$
- ▶ W'keit dass 1. und 2. besetzt:  $P_2 = \frac{M}{N} \cdot \frac{M-1}{N-1} \approx \alpha \cdot \alpha = \alpha^2$
- ▶ W'keit dass 1. und 2. und 3. besetzt:  $P_3 \approx \frac{M}{N} \cdot \frac{M-1}{N-1} \cdot \frac{M-2}{N-2} \approx \alpha^3$
- ▶ ...
- ▶ W'keit für  $k$  erfolglose Sondierungen (alles besetzt):  $P_k \approx \alpha^k$ .



## Sondieren: Aufwandsbetrachtung (cont'd)



#Sondierungen im erwarteten Mittel:

$$= 1 + \alpha \cdot 1 + \alpha^2 \cdot 1 + \alpha^3 \cdot 1 \dots + \alpha^{N-1} \cdot 1$$

in 100%  
die 1. Stelle  
sondieren

in  $\alpha$  aller  
Fälle auch  
die 2. Stelle  
sondieren

in  $\alpha^2$  aller  
Fälle auch  
die 3. Stelle  
auschauen

letzte  
Sondierung

$$= 1 + \alpha + \alpha^2 + \dots + \alpha^{N-1} \approx 1 + \alpha + \alpha^2 + \dots = \frac{1}{1 - \alpha}$$

(keine Obergrenze  
→ geometrische Reihe)

# Sondieren: Aufwandsbetrachtung (cont'd)

## Ergebnis

- Die Anzahl der Sondierungen ist der **Haupttreiber** des Aufwands: Die anderen Schritte (*z.B.  $h$  berechnen*) sind  **$O(1)$** .
- Wir brauchen im Schnitt  $\approx 1/(1 - \alpha)$  Sondierungen.
- **Wichtigster Parameter: Füllgrad  $\alpha$ .**
- Ist der Füllgrad nicht zu hoch (< 80%), ist Hashing mit Sondierung im Durchschnitt (*auch in der Praxis*) **performant**.

Füllgrad $\alpha$	# Sondierungen		
	Theoretische Betrachtung $(1 / (1 - \alpha))$	Praxistest	
		linear	quadratisch
50%	2.0		
70%	3.3		
80%	5.0		
90%	10.0		
95%	20.0	200	22
99%	100.0		

## Sondieren: Bewertung

- ☺ kein **zusätzlicher Speicher** nötig, nur die Hash-Tabelle.
- ☺ in der **Praxis**: gute Laufzeiteigenschaften.
  
- ☹ **Entartung/Klumpenbildung** möglich, schwer abzufangen.
- ☹ empfindlich bei **hohem Füllgrad**.
- ☹ **Löschen** sehr aufwändig  
(oft werden *Elemente nur als gelöscht markiert* ).



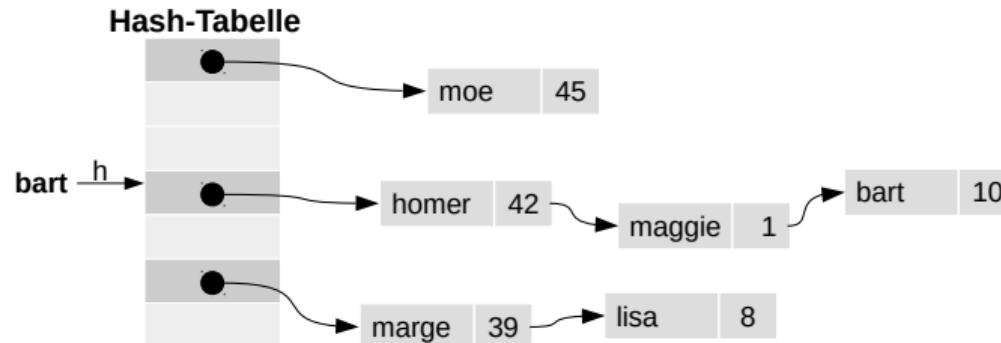


# Outline

1. Hash-Funktionen
2. Kollisionsbehandlung 1: Sondieren
3. Kollisionsbehandlung 2: Verkettung

# Verkettung

Die zweite Behandlung von Kollisionen lautet **Verkettung**.

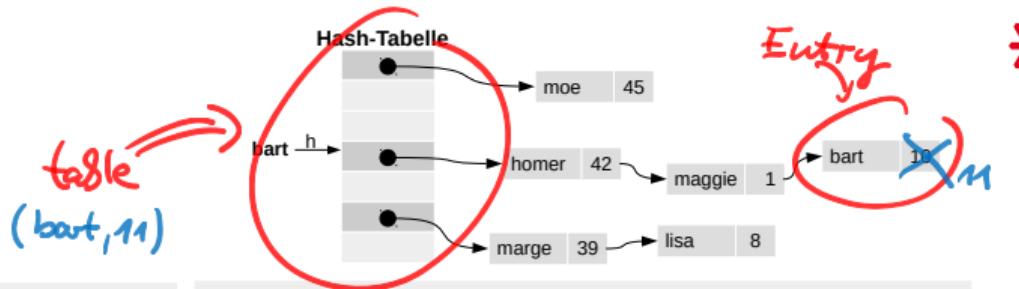


## Ansatz

- Die Schlüssel-Wert-Paare befinden sich nicht in der **Tabelle selbst**, sondern in zusätzlichen **Containern** (hier: **Listen**).
- Die Schlüssel mit **gleichem Hash-Wert** “**teilen**” sich eine Liste → **keine Kollisionen**.
- **Suchen/Einfügen/Löschen**: Nachschlagen Hash-Wert, dann Operationen der entsprechenden Liste.

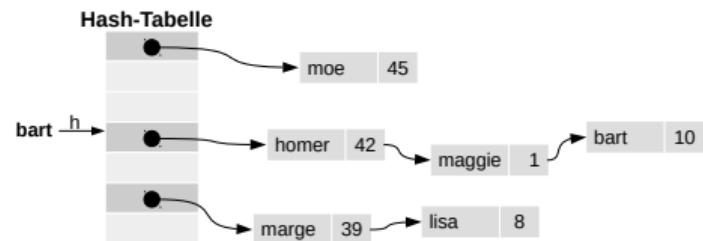
# Verkettung: Implementierung

```
public class HashMap<K,V> {  
    private class Entry {  
        K key;  
        V value;  
    }  
    int hashCode(K key) {...}  
    LinkedList<Entry>[] table;
```



```
public void add(K key, V value) {  
    int pos = hashCode(key);  
    LinkedList<Entry> l = table[pos];  
    if (l == null)  
        l = table[pos] = new LinkedList();  
    Entry e = l.find(key);  
    if (e != null) // Schlüssel vorhanden  
        e.value = value;  
    else // Schlüssel nicht "  
        l.addFirst(new Entry(key, value));  
}
```

## Verkettung: Implementierung (cont'd)



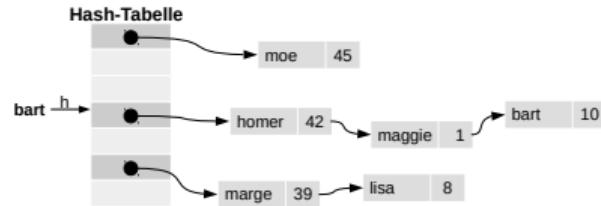
# Verkettung: Aufwand

## Annahmen

- ▶ **Perfekte Hash-Funktion:** Hash-Werte sind **gleichverteilt** über die Tabelle.
- ▶  $\alpha = M/N$  sei (*immer noch*) der **Füllgrad** (*eine Überbelegung, d.h.  $M > N$ , ist möglich!*).

## Aufwandsbetrachtung

- ▶ Entscheidend ist die **Listenlänge!**
- ▶ **Durchschnittliche Länge** einer Liste ist  $M/N = \alpha$ .
- ▶ **Worst Case (erfolglose Suche):**  $1 + \alpha$  Sprünge  
(*einen in die Hash-Tabelle, und  $\alpha$  durch die Liste*).
- ▶ **Average Case (erfolgreiche Suche):**  $1 + \alpha/2$  Sprünge  
(*im Durchschnitt bis zur Mitte der Liste*).
- ▶ **Sehr gute** Laufzeiteigenschaft! Quasi kontant bei nicht überbelegter Tabelle.



Füllgrad $\alpha$	Sprünge
50%	1.25
90%	1.45
100%	1.5
200%	2
1000%	6

# Verkettung: Bewertung

- ☺ Konzeptuell **einfacher** als Sondierung.
- ☺ **Löschen** einfach möglich.
- ☺ sehr gute Laufzeit, auch bei **hohem Füllgrad**.
- ☹ Benötigt **viel Speicher**:  
(1) Löcher in Tabelle, (2) extra Listen
- ☹ **Entartung** möglich (*siehe Sondierung*). Alle Werte  
in derselben Liste → Einfügen, Suche, Löschen sind  $\Theta(n)$ !
- ☺ Können Entartung entgegenwirken, indem wir Listen durch **bessere Container**  
(z.B. *balancierte Bäume*,  $O(\log n)$ ) ersetzen.  
In der Praxis ist dies in der Regel nicht nötig.



## References I

