

# Algorithmen und Datenstrukturen

– Wintersemester 2019 –

## Kapitel 08: Hashing

Fachbereich Informatik  
TU Kaiserslautern

Dozent: Dr. Patrick Michel

Folien ursprünglich von Prof. Dr. Adrian Ulges (Hochschule RheinMain)

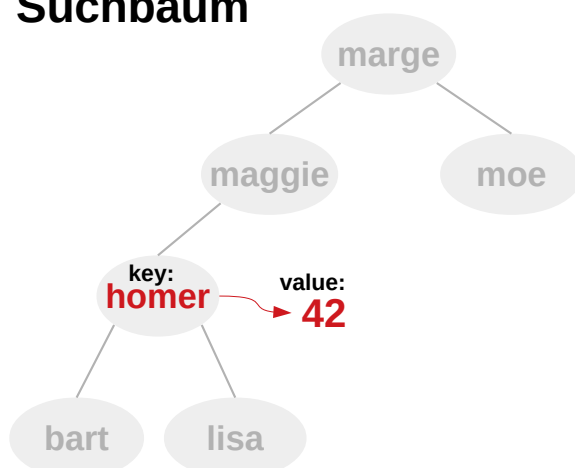
1

### Hashing

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

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

#### Suchbaum



#### Hashing

key	value
marge	39
bart	42
moe	45
lisa	8
<b>homer</b>	<b>42</b>
maggie	1

2



# 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

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

Beispiel (rechts)

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

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

5

## Outline

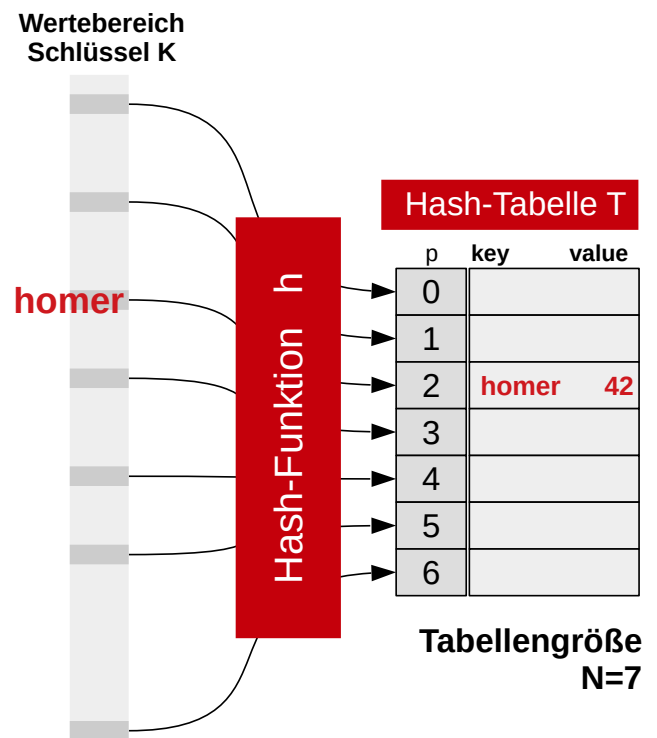
1. Hash-Funktionen
2. Kollisionsbehandlung 1: Sondieren
3. Kollisionsbehandlung 2: Verkettung

6

# 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**  $h(k) \in \{0, 1, \dots, N-1\}$  in der Hash-Tabelle.

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  $k$ .

8

# Hashing: Naive Implementierung

- ▶ 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.

```
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;  
    }  
}
```

9

## Problem: Kollisionen

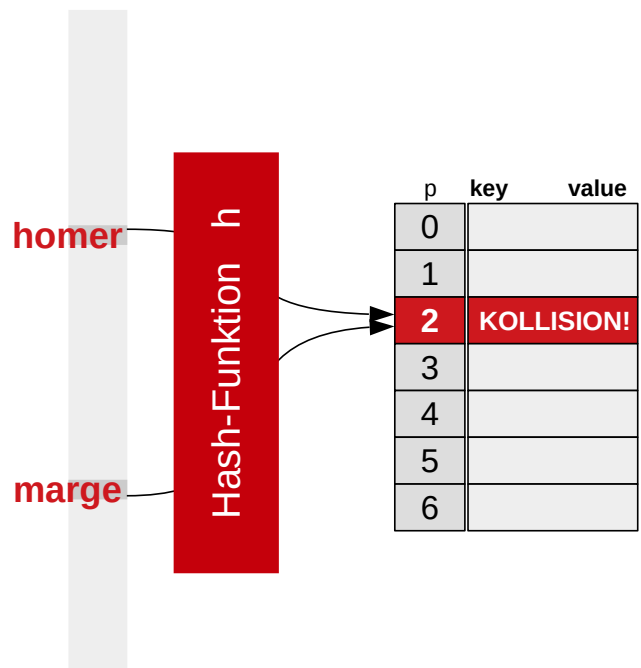
- ▶ Meist gilt  $N \ll \#K$   
(die Tabelle ist deutlich kleiner als die Schlüsselmenge).

- ▶ Gemäß dem **Taubenschlagprinzip** folgt:

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

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

Wertebereich  
Schlüssel K

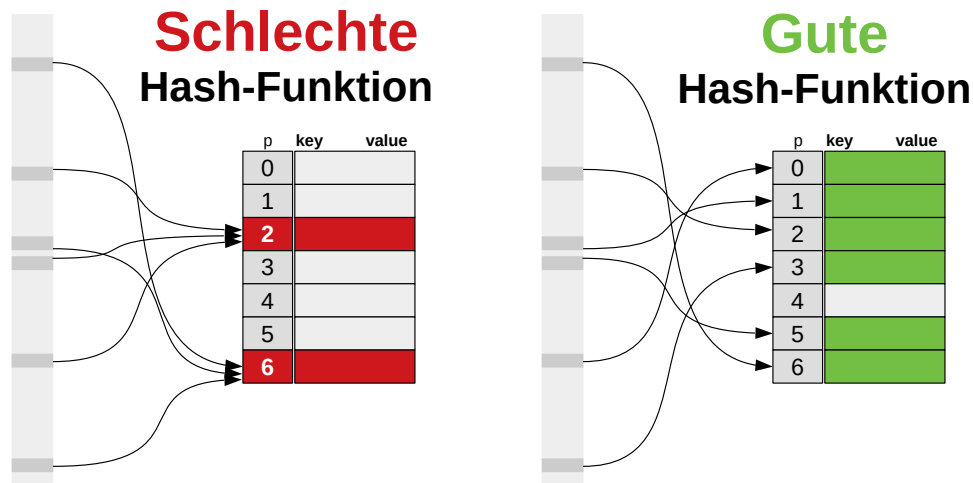


# Strategien gegen Kollisionen

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

“Gute” Hash-Funktionen...

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



11

## Hash-Funktionen für Strings: Beispiele

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

1. **int-Schlüssel**

$$h(k) = k/N$$

2. **int-Schlüssel, Version 2**

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

12

# 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, restliche Bits werden ignoriert.

## Gute Werte für *N* ...

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

13

## 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:  
$$h(k) = (k_1 + k_2 + k_3) \% N$$

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

14

# 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

15

## 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:

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

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

### Beispiel

Basis  $B=10$ , String  $k="ADS"$ .

$$h("ADS") = \left( A \cdot 100 + D \cdot 10 + S \cdot 1 \right) \% N$$

16



## 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. `Java's String.hashCode()`).

### Problem

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

$$\begin{aligned} h(\text{"With power comes great responsibility"}) \\ &= \mathbf{W} \cdot 10^{36} + i \cdot 10^{35} + t \cdot 10^{34} \dots \\ &= \mathbf{57000000000000000000000000000000000000} + \dots \end{aligned}$$

17

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

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

$$k_1 \cdot B^{n-1} + k_2 \cdot B^{n-2} + \dots + k_{n-1} \cdot B^1 + k_n \cdot B^0$$

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

18

# 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 + si ) % N // Überlauf vermeiden
return h
```

19

## Beispiel

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

p	key	value
0		
1		
2		
3		
4		
5		
6		

20

# Outline

## 1. Hash-Funktionen

## 2. Kollisionsbehandlung 1: Sondieren

## 3. Kollisionsbehandlung 2: Verkettung

21

## Sondieren

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

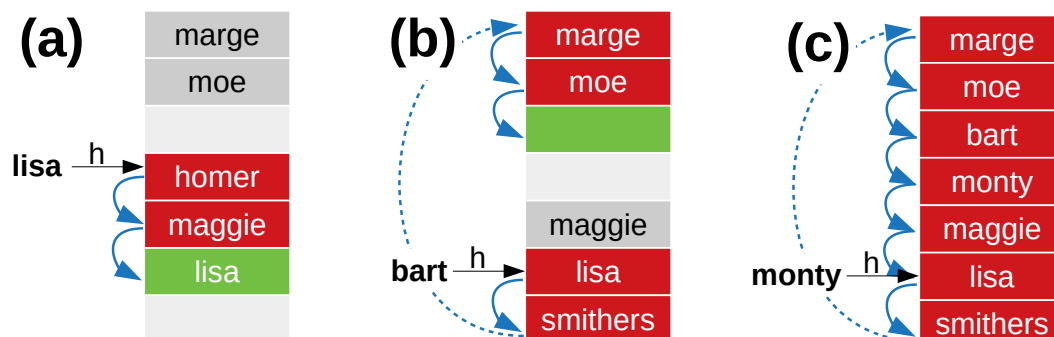
### Lösung 1: Sondieren

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.



22

# Sondieren: Generell

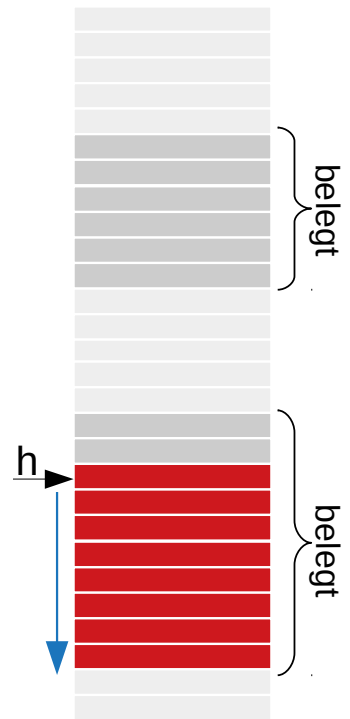
## Problem: Klumpenbildung

- ▶ 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**:  $g(m) = (h(k) + m) \% N$ .



23

# Weitere Sondierungsfunktionen

## Linear, größere Schritte

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

## Quadratisches Sondieren

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

24

# 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 (?)

## (a) linear

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

## (b) quadratisch

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

25

# 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 ).



26

# Outline

## 1. Hash-Funktionen

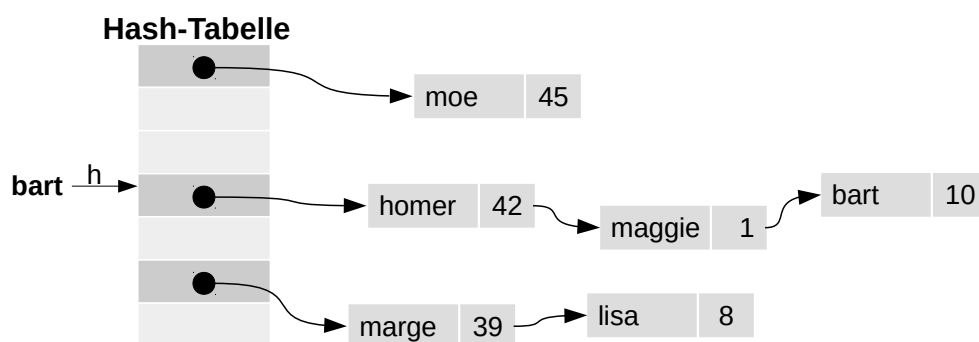
## 2. Kollisionsbehandlung 1: Sondieren

## 3. Kollisionsbehandlung 2: Verkettung

27

## Verkettung

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

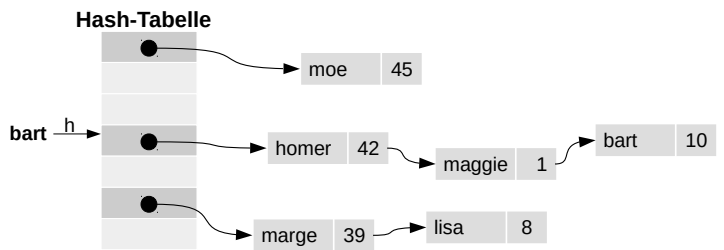


### Ansatz

- ▶ 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.

28

# Verkettung: Implementierung



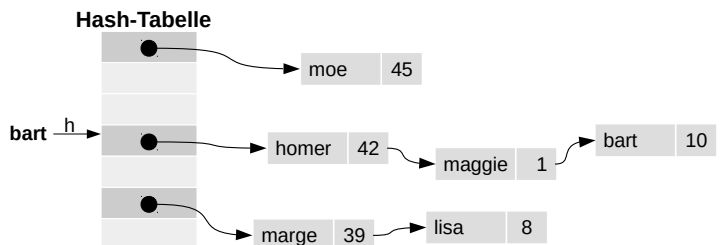
```
public class HashMap<K,V> {  
    private class Entry {  
        K key;  
        V value;  
    }  
}
```

29

# Verkettung: Aufwand

## Annahmen

- ▶ **Perfekte Hash-Funktion:** Hash-Werte sind **gleichverteilt** über die Tabelle.
- ▶  $\alpha = M/N$  sei (immer noch) der **Füllgrad**.



## Entscheidender Faktor: Listenlänge

- ▶ **Durchschnittliche Länge** einer Liste ist  $M/N = \alpha$ .
- ▶ Bei der **erfolglosen Suche** benötigen wir also  $1 + \alpha$  Sprünge (einen in die Hash-Tabelle, und  $\alpha$  durch die Liste).
- ▶ Bei einer **erfolgreichen Suche** benötigen wir  $1 + \alpha/2$  Sprünge (im Durchschnitt bis zur Mitte der Liste).
- ▶ **Sehr gute** Laufzeiteigenschaft! Quasi konstant bei nicht überbelegter Tabelle.

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

30

# 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 einer der Listen → Einfügen, Suche, Löschen sind  $O(n)$ !
- ☺ Können Entartung entgegenwirken, indem wir Listen durch **bessere Container** (z.B. *balancierte Bäume*,  $O(\log n)$ ) ersetzen. In der Praxis ist dies meist nicht nötig.



## References I