DHBW
Duale Hochschule
Baden-Württemberg
Stuttgart
Campus Horb

Algorithmen und Datenstrukturen

Teil 12:
Suchen (III) - Hashing

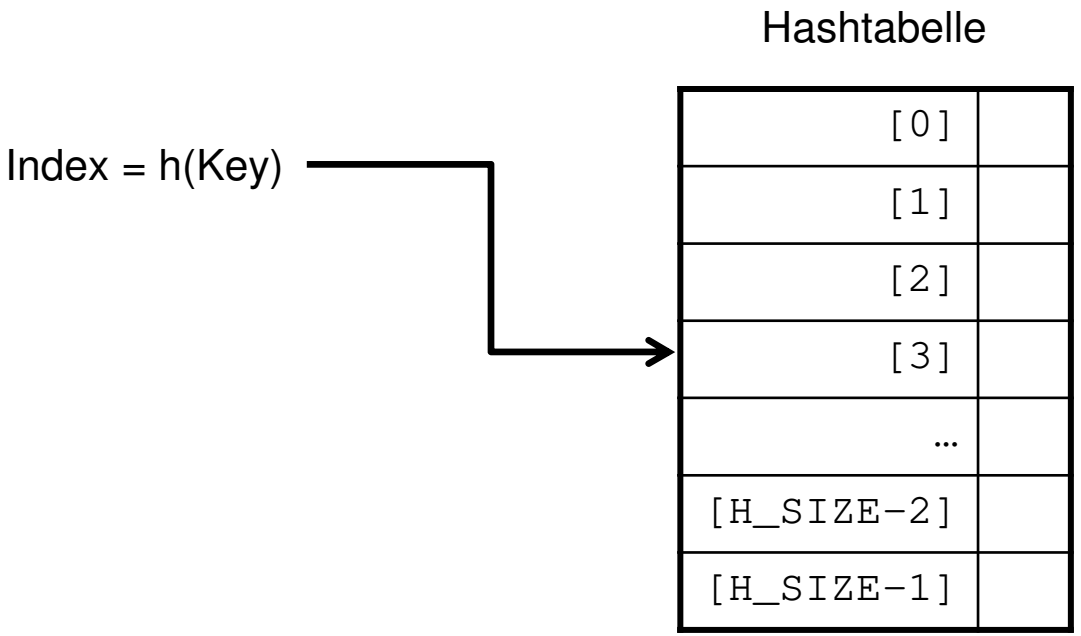
DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Olaf Herden
Stand: 06/2020

Gliederung

- Grundbegriffe
- Kollisionsbehandlung
- Suchen und Löschen
- Bewertung

Hashing/Hash-Funktion/Hash-Tabelle

- Hashing (to hash = zerhacken): Abbildung von Ausgangsmenge auf Zielmenge
- Abbildung h (Hash-Funktion)
- h(k) Hash-Wert (Synonym: Hash-Code, Index) von Schlüssel k
- Zugrundeliegende Datenstruktur wird als Hash-Tabelle H bezeichnet
- H habe Größe `H_SIZE`



Perfektes Hashing und Kollisionen

- Perfektes Hashing:
  Für alle Schlüssel k_1, k_2 mit $k_1 \neq k_2$ gilt stets $h(k_1) \neq h(k_2)$
- Kollision: $h(k_1) = h(k_2)$ für $k_1 \neq k_2$
- Perfektes Hashing kann nicht immer garantiert werden
- Daher notwendig: Kollisionsbehandlung
  - Verkette Überläufer (Chaining, Open Hashing)
  - Sondierung (Probing, Open Adressing, Closed Hashing):
    - Suche alternative Position
    - Strategien:
      - Lineares Sondieren
      - Quadratisches Sondieren
      - …
  - Überlaufbereich (Overflow Area):
    - Speicherung kollidierender Elemente in separatem Bereich

Hash-Funktion: Eigenschaften (I)

- Eindeutigkeit/Reproduzierbarkeit:
  - Funktion muss eindeutig von Quellmenge auf Zielmenge abbilden
  - Wiederholtes Berechnen des Hash-Wertes desselben Quellelements muss dasselbe Ergebnis liefern
- Unumkehrbarkeit:
  - Zur Hash-Funktion gibt es keine inverse Funktion, mit der es möglich wäre, für ein gegebenes Zielelement ein passendes Quellelement zu berechnen
- Datenreduktion:
  - Speicherbedarf Hash-Wert muss deutlich kleiner sein als Eingabewert
- Zufälligkeit:
  - Ähnliche Quellelemente sollten möglichst zu völlig verschiedenen Hash-Werten führen
- Gute Verteilung:
  - Hash-Werte sollten möglichst gut über Zielmenge verteilt sein

Hash-Funktion: Eigenschaften (II)

- Kollisionsfreiheit/-armut:
  - Optimal ist Kollisionsfreiheit, d.h. zwei verschiedene Quellelemente haben stets unterschiedliche Hash-Werte („Perfektes Hashing“)
  - I.A. nicht garantierbar, daher schwächere Eigenschaft Kollisionsarmut, d.h. seltenes Auftreten von Kollisionen
- Effizienz:
  - Hash-Funktion muss schnell berechenbar sein

Hash-Funktion: Beispiel (I)

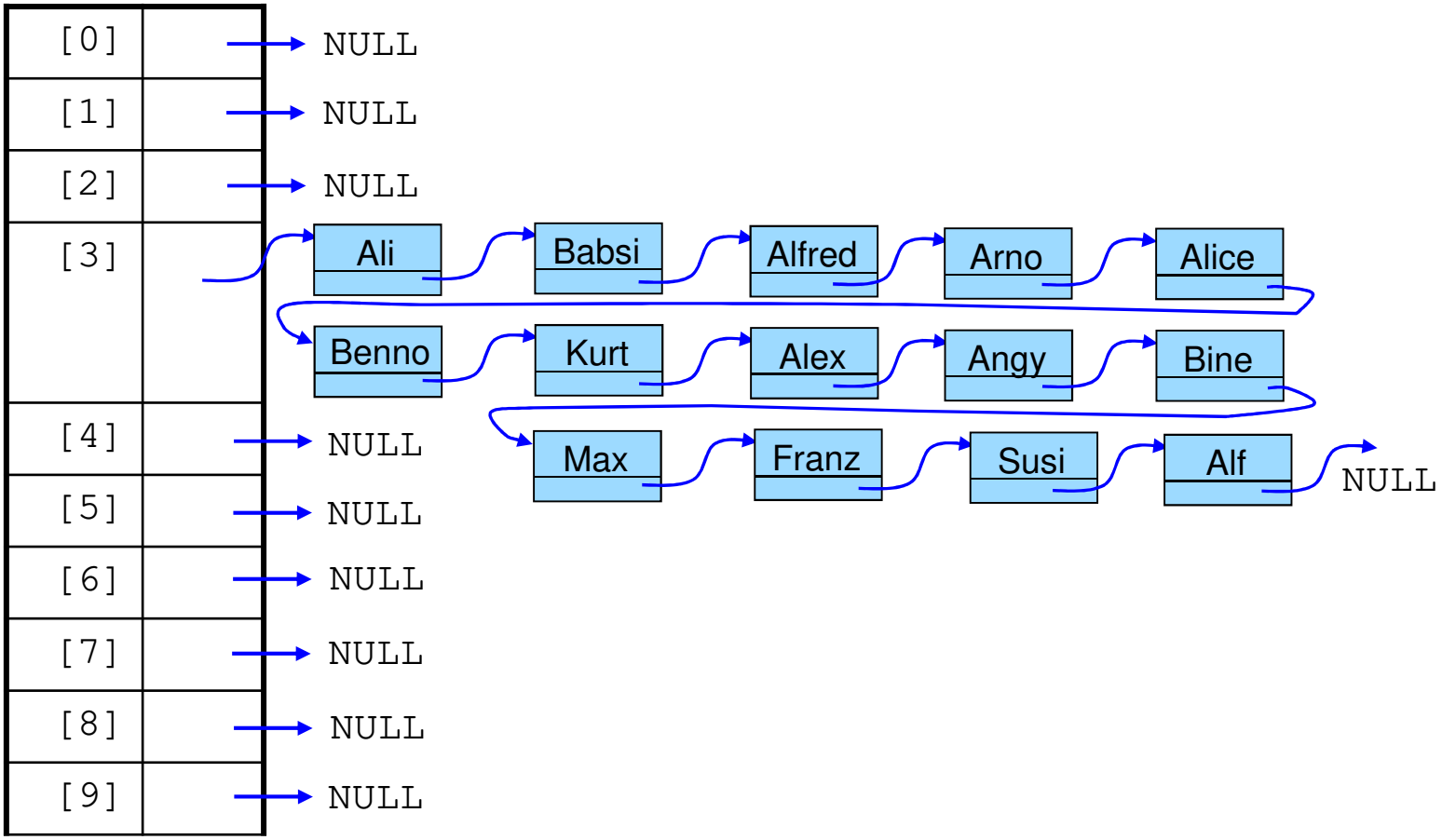
- Szenario:
  - Ausgangsdaten: Menge der Vornamen
  - Zielmenge: Werte von 0 bis 9
  - Enfügen von "Ali", "Babsy", "Alfred", "Arno", "Alice", "Benno", "Kurt", "Alex", "Angy", "Bine", "Max", "Franz", "Susi", "Alf" in Hash-Tabelle
  - Kollisionsbehandlung: Verkettete Überläufer
  - Betrachtung verschiedener Hash-Funktionen
  - Maß für Bewertung: Summe Suchschritte für jedes einzelne Element
  - Optimales Maß: 14 (ein Suchschritt pro Element)
- Beispiel stammt von http://www.tfh-berlin.de/~oo-plug/Ad/Hash.html (Seite nicht mehr verfügbar)

Hash-Funktion: Beispiel (II)

- Variante 1 (Konstante Hash-Funktion)

```
int hash_1(String s) {return 3;}
```

Resultat:



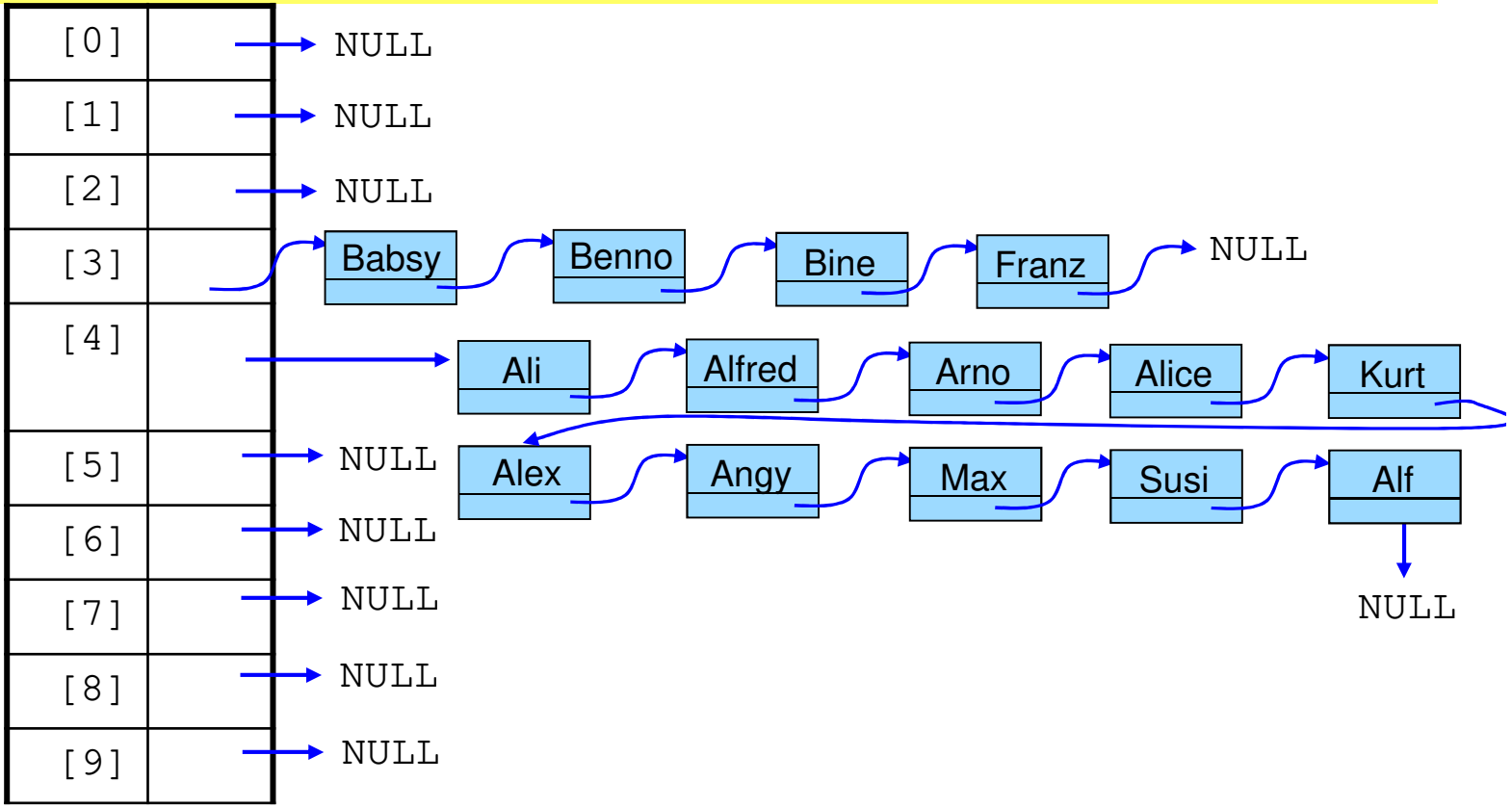
- Bewertung: Sehr schlechte Hash-Funktion (105 Suchschritte), entspricht Liste

Hash-Funktion: Beispiel (III)

- Variante 2 (Erstes Zeichen wird gerade/ungerade codiert)

```
int hash_2(String s) {if (s.charAt(0) % 2 == 0) return 3;
                      else return 4;}
```

Resultat:



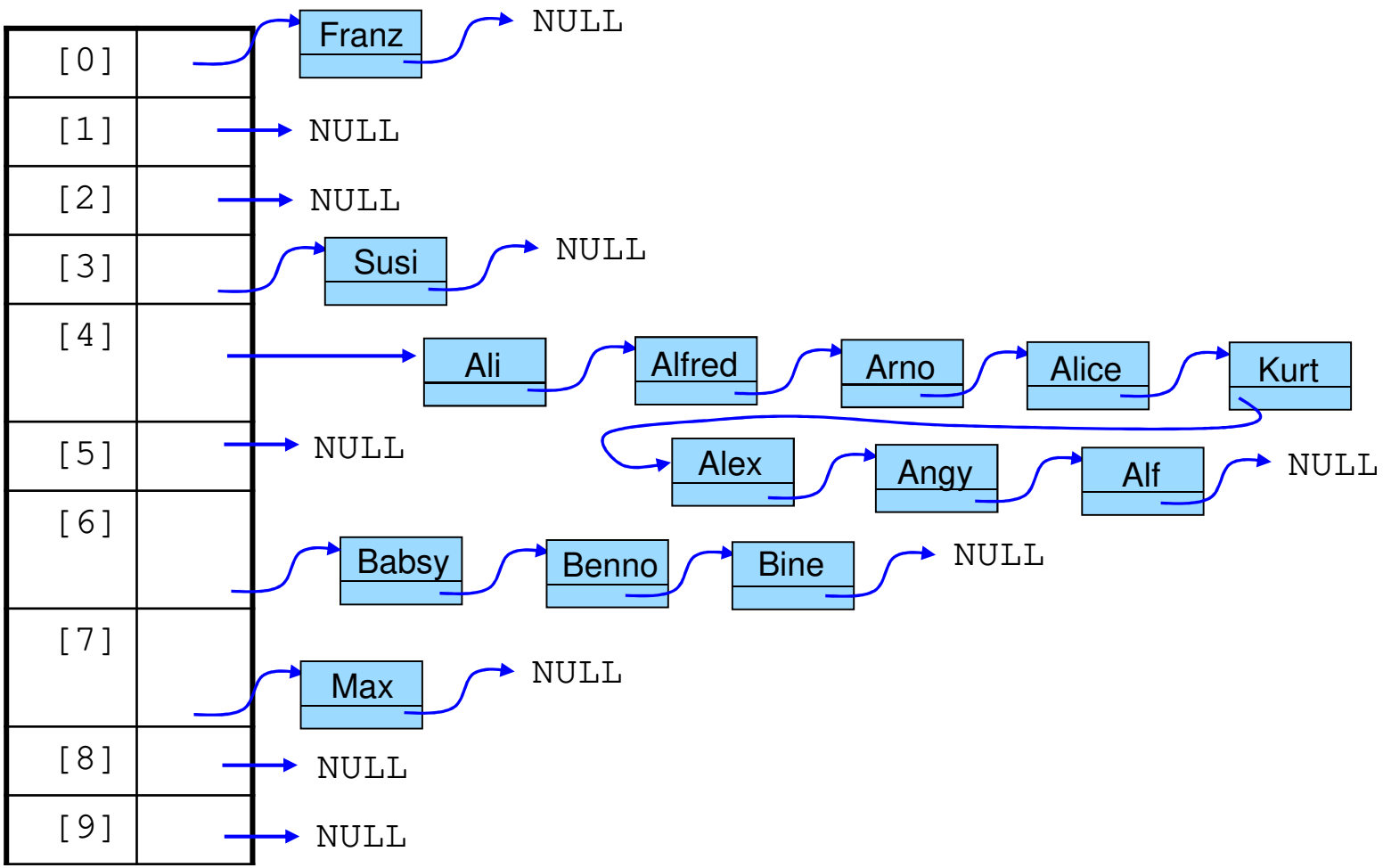
- Bewertung: Etwas besser als erste Variante (65 Suchschritte), aber ein Großteil der Zielmenge bleibt nach wie vor ungenutzt

Hash-Funktion: Beispiel (IV)

- Variante 3 (Berechnung aus erstem Zeichen)

```
int hash_3(String s) {return s.charAt(0) % H_SIZE;}
```

Resultat:



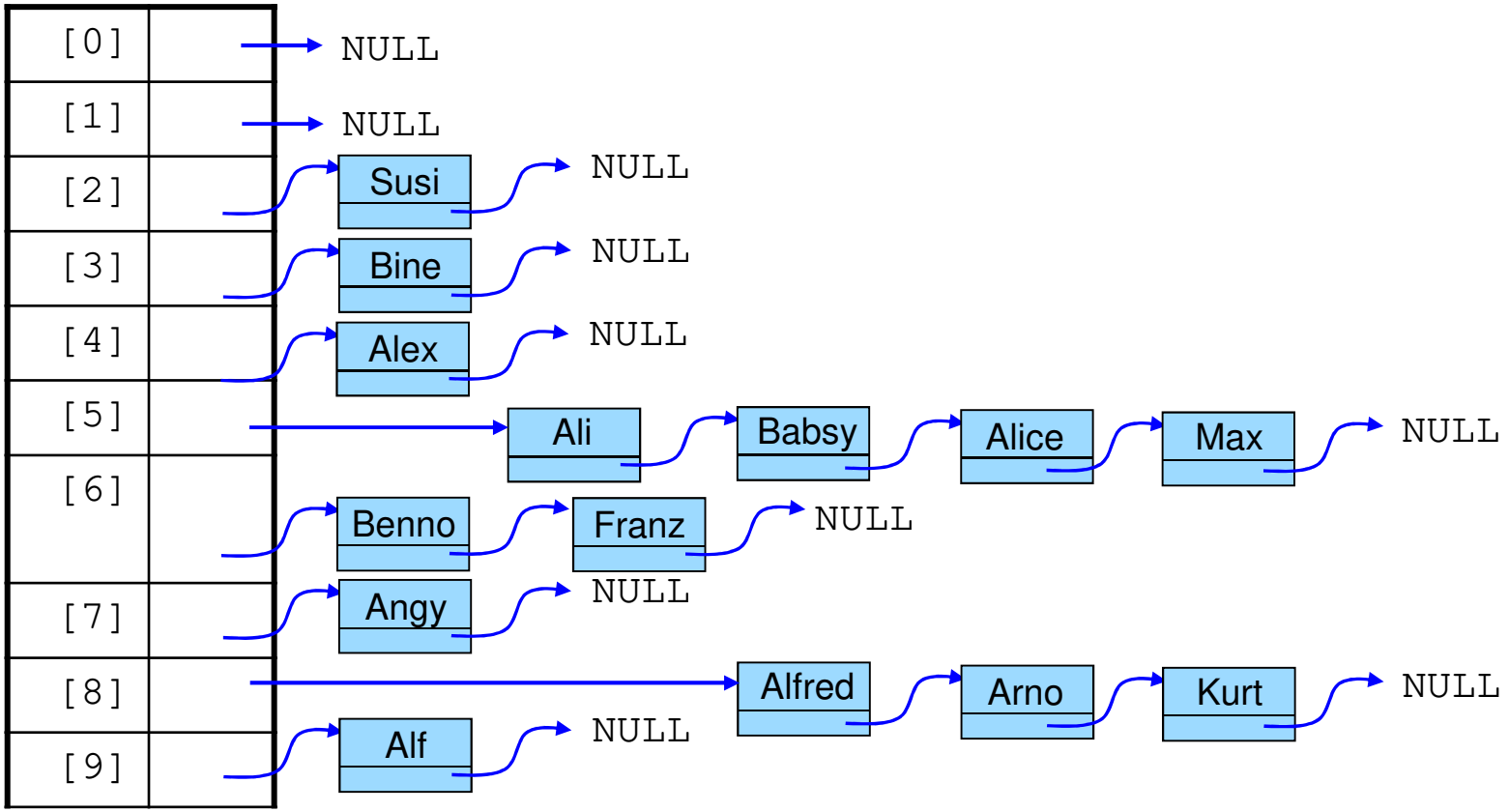
- Bewertung: Besser (45 Suchschritte), aber: viele Listen bleiben leer

Hash-Funktion: Beispiel (V)

- Variante 4 (Berechnung aus drei Zeichen)

```
(1) int hash_4(String s){
(2)  int first = s.charAt(0) % 3;
(3)  int middle = s.charAt(s.size()/2) % 5;
(4)  int last = s.charAt(s.size()-1) % 7;
(5)  return (first+middle+last) % H_SIZE;}
```

Resultat:



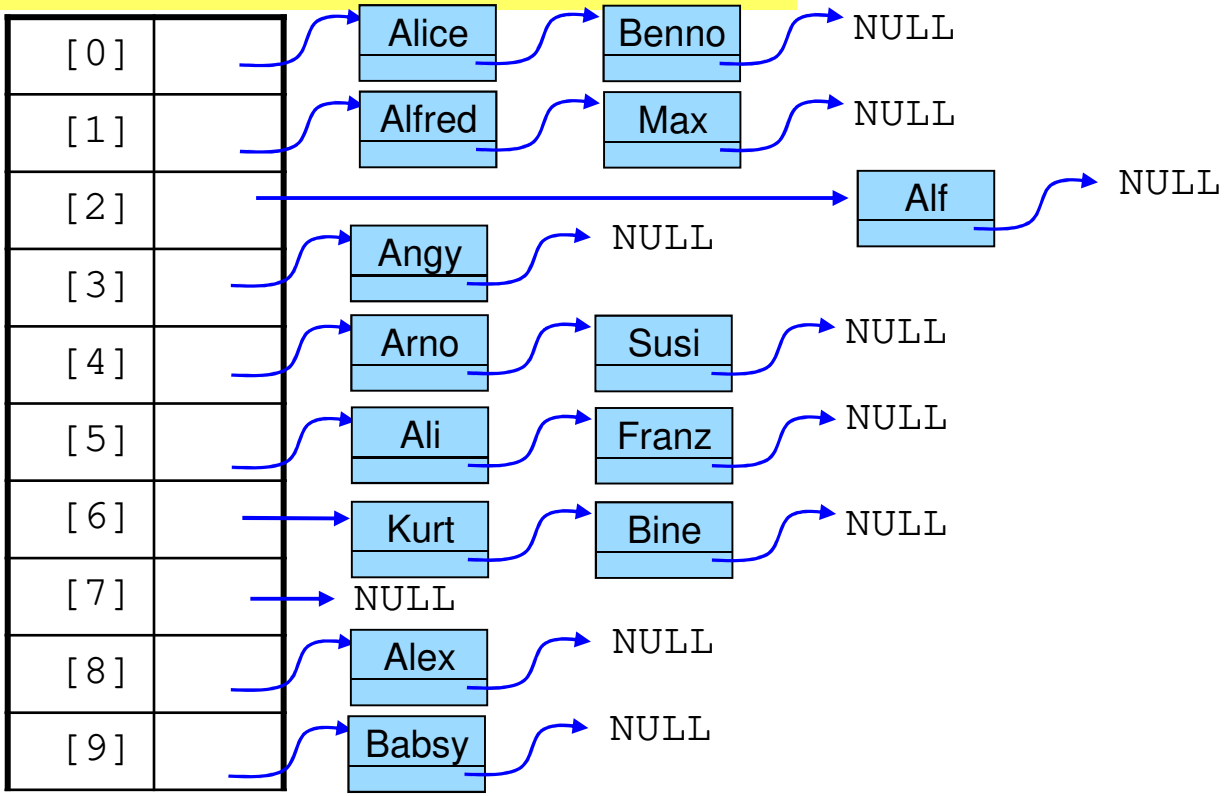
- Bewertung: Recht gute Verteilung (24 Suchschritte)

Hash-Funktion: Beispiel (VI)

- Variante 5 (Berücksichtigung aller Zeichen)

```
(1) int hash_5(String s){
(2)   int i = 0;
(3)   for (int j=0; j<s.size(); j++)
(4)     i += s[j] % 32 + j;
(5)  return i % H_SIZE;}
```

Resultat:



- Bewertung: Nahezu optimale Verteilung (19 Suchschritte)

Hash-Funktionen: Datentypen

- Hash-Funktion hängt von Datentyp und konkreter Anwendung ab
- Ganzzahlige Werte: Direkte Eingabe in Hash-Funktion
- Andere Datentypen:
  - Rückführung auf ganze Zahlen:
    - Fließkommazahlen z.B. Addition von Mantisse und Exponent
    - Zeichenketten z.B. Addition der ASCII- oder Unicode-Werte ausgewählter Zeichen

- Prinzipielle Konzepte für Hash-Funktionen:
  - Divisionsmethode
  - Folding
  - Mid-Square-Methode

Divisionsmethode

- Bilde ganzzahligen Rest der Division durch Größe Hash-Tabelle
- $h(k) = k \; mod \; N$
- Funktioniert am besten (Verteilung), wenn N Primzahl
- Beispiel:
  - Hash-Tabelle mit $H\_SIZE = 13$
  - Einfügen 55.456
  - $h(55.456) = 55.456 \; MOD \; 13 = 11$

Folding

- Zerlege k in gleich große Teile (außer letztem Teil)
- Addiere diese Teile unter Vernachlässigung des letzten Übertrag
- Beispiel:
  - Hash-Tabelle Kapazität 1000
  - Zerlege k in Teil der Länge 3 (+Rest) und bilde Summe
  - Beispiel: k = 53746312526

537	463	125	26

```
   537
+  463
+  125
+   26
------
  1151
```

  - h(53726312526) = 151

Mid-Square-Methode

- Bilde Quadrat von k
- Nimm mittleren Teil hiervon als Hash-Wert
- Beispiel:
  - Hash-Tabelle $H\_SIZE = 1000$
  - Hash-Wert ab Position p mit $p = Länge\ k^2\ DIV\ 2$
  - Beispiel:
    - $k = 102.726$
    - $k^2 = 10.552.631.080$
    - $p = 11\ DIV\ 2 = 5$

      10.552.631.080

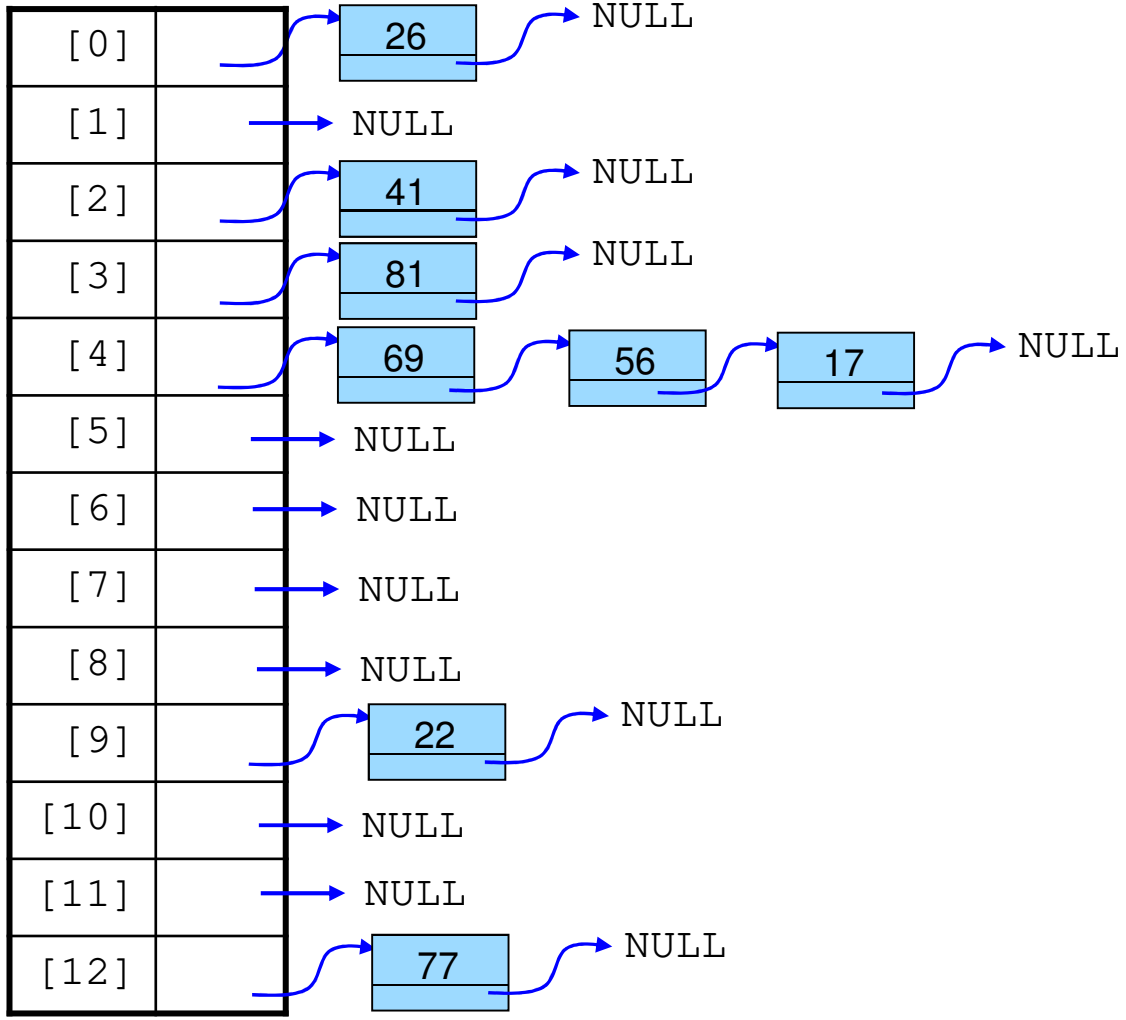
    - $h(102.726) = 263$

Übersicht

- Grundbegriffe
- Kollisionsbehandlung
- Suchen und Löschen
- Bewertung

Verkettete Überläufer (I)

- Realisierung Hash-Tabelle als Feld von Listen
- Synonym: Chaining, Open Hashing
- Beispiel:
  - $h(k) = k \; MOD \; 13$
  - Füge Werte 22, 41, 69, 26, 56, 17, 77 und 81 ein

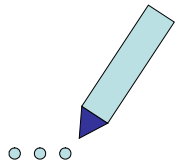


Verkettete Überläufer (II)

- Handhabung langer Listen:
  - Zielbereich Hash-Funktion zu klein ⇒ Effizienz kann nachlassen (lange Listen müssen durchsucht werden)
  - Vorgehensweise:
    - Überschreiten Listenlängen gewisse Schwellwerte, wird dynamisch „nachgebessert“
    - Geschieht durch Vergrößerung der Zielmenge
    - Alle Werte müssen per Hash-Funktion neu zugeordnet werden

Lineares Sondieren (I): Prinzip

- Sei Q Ausgangsmenge, T Hash-Tabelle, h Hash-Funktion
- Einfügen eines neuen Wertes k $\in$ Q:
  - Wenn $T[h(k)]$ frei, dann füge k ein
  - Sonst: Versuche $T[h(k)+i]$ für $i = 1, 2, 3, ...$
- Beispiel:
  - Hash-Funktion $h(k) = k\ MOD\ 13$
  - Füge nacheinander Werte 22, 41, 69, 26, 56, 17, 77 und 81 ein

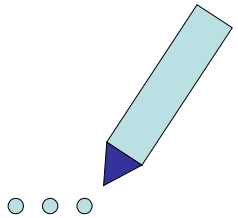


Lineares Sondieren (VI): Anmerkungen

- Sekundärkollision: Kollision, die erst beim Sondieren auftritt
- Lineares Sondieren wird auch in folgenden Varianten verwendet:
  - Versuche $T[h(k) + i \cdot c]$ für $i = 1, 2, 3, \ldots$ und eine Konstante c
  - Versuche $T[h(k) + i], T[h(k) - i]$ für $i = 1, 2, 3, \ldots$

Quadratisches Sondieren (I): Prinzip

- Sei Q Ausgangsmenge, T Hash-Tabelle, h Hash-Funktion
- Einfügen eines neuen Wertes k $\in$ Q:
  - Wenn $T[h(k)]$ frei, dann füge k ein
  - Sonst: Versuche $T[h(k) + i^2]$ für $i = 1, 2, 3, \ldots$
- Beispiel:
  - Hash-Funktion $h(k) = k\ MOD\ 13$
  - Füge nacheinander Werte 22, 41, 69, 26, 56, 17, 77 und 81 ein



Überlaufbereich

- Speichere kollidierende Elemente in separatem Bereich
- Beispiel:

[0]	26
[1]	
[2]	41
[3]	81
[4]	69
[5]	
[6]	
[7]	
[8]	
[9]	22
[10]	
[11]	
[12]	77

Überlaufbereich:

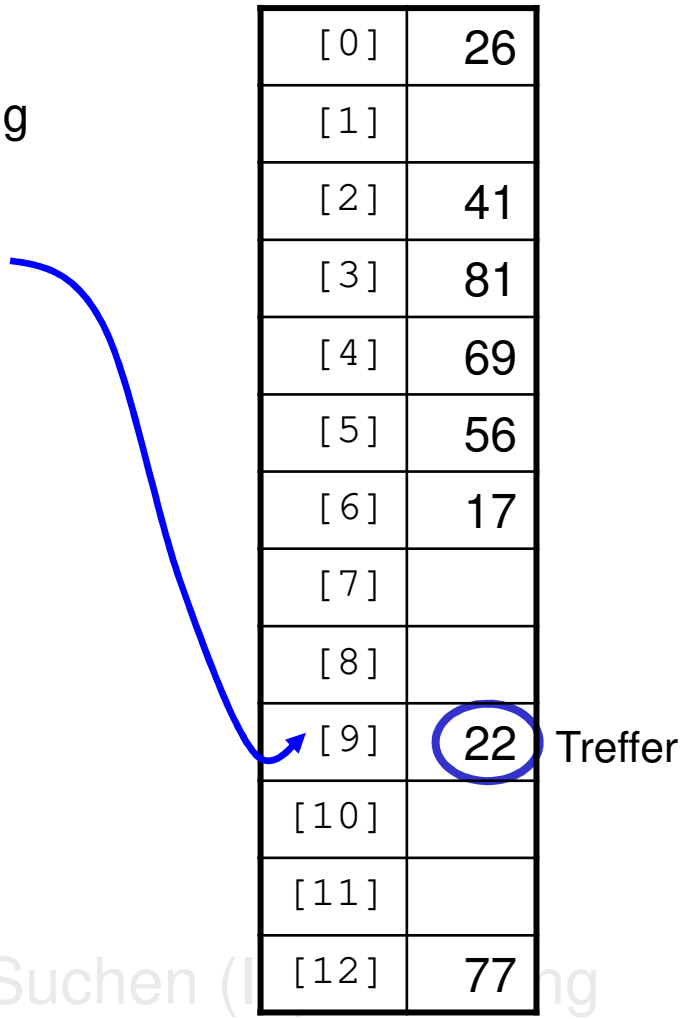
[0]	56
[1]	17
[2]	
[3]	
[4]	

Übersicht

- Grundbegriffe
- Kollisionsbehandlung
- Suchen und Löschen
- Bewertung

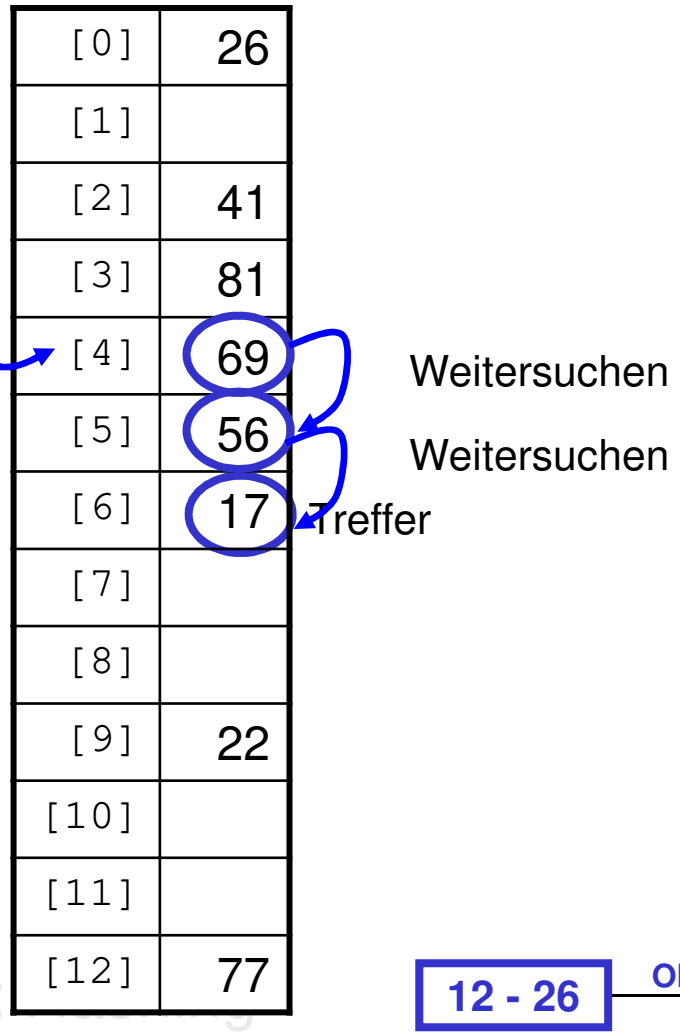
Suchen (I)

- Berechnung Hash-Wert des gesuchten Elements
- Zugriff auf Hash-Tabelle
- Evtl. Weitersuchen gemäß Sondierungsstrategie
- Beispiel:
  - Ann.: Lineare Sondierung
  - Suchen von 22
  - $h(22) = 22\ MOD\ 13 = 9$



Suchen (II)

- Beispiel:
  - Ann.: Lineare Sondierung
  - Suchen von 17
  - $h(17) = 17\ MOD\ 13 = 4$



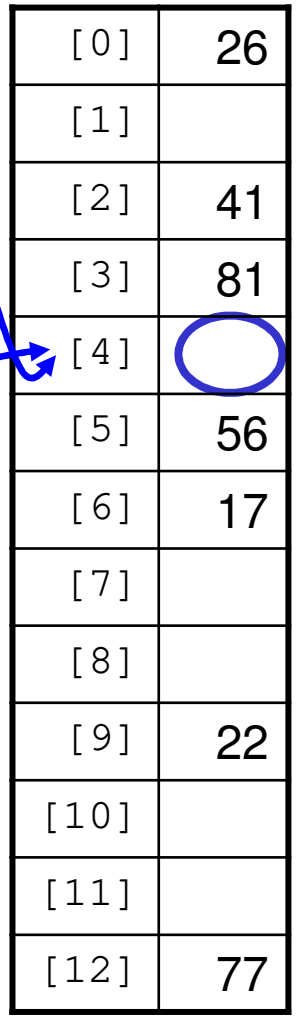
- Bezeichnung: Pfad von ursprünglichem Hash-Wert über Sondierungswerte nennt man Sondierungskette

Löschen (I)

- Vorgehen scheint zunächst offensichtlich:
  - Suchen (gemäß Hash-Funktion und Sondierungsstrategie) und Entfernen zu löschender Eintrag
  - Funktioniert bei verkettetem Überläufer
  - Mögliches Problem bei Sondierungen:
    - Auf zu löschenden Eintrag folgen weitere Einträge, die aufgrund von Sondierungen dahin gelangt sind und nun nicht mehr gefunden werden
  - Beispiel: Resultat von vorhin mit linearem Sondieren

Löschen (II): Beispiel

- Löschen von 69, Suchen von 17:
  - $h(69) = 69\ MOD\ 13 = 4$
  - $h(17) = 17\ MOD\ 13 = 4$



[0]	26
[1]	
[2]	41
[3]	81
[4]	
[5]	56
[6]	17
[7]	
[8]	
[9]	22
[10]	
[11]	
[12]	77

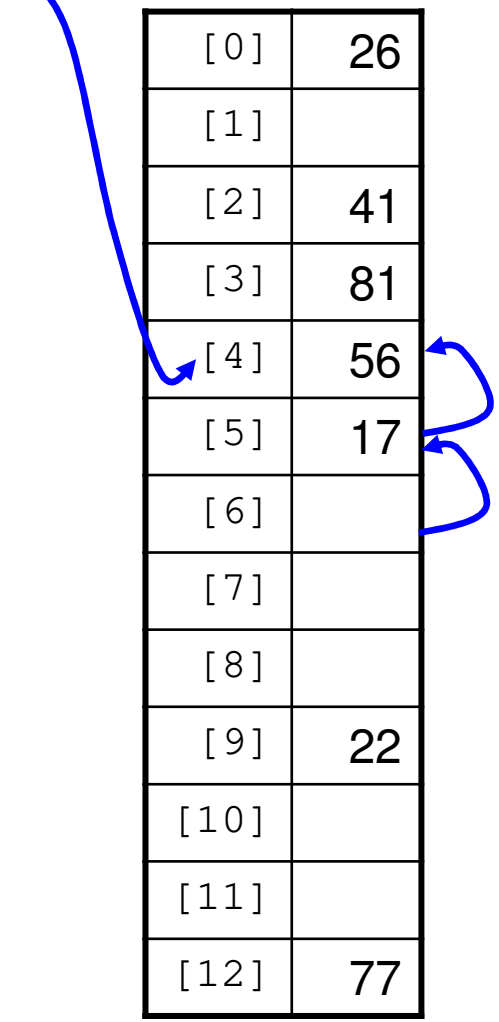
Kein Treffer, obwohl Element in Hash-Tabelle

Löschen (III)

- Strategien zur Vermeidung von Unterbrechungen in Sondierkette:
  - Reorganisation:
    - Alle in Frage kommenden weiteren Einträge werden überprüft und gegebenenfalls verschoben
  - Markierung:
    - Markierung gelöschter Eintrag mit Status „Gelöscht“
    - Hier können neue Werte eingefügt werden
    - Keine Unterbrechung Sondierkette

Löschen (IV): Reorganisation

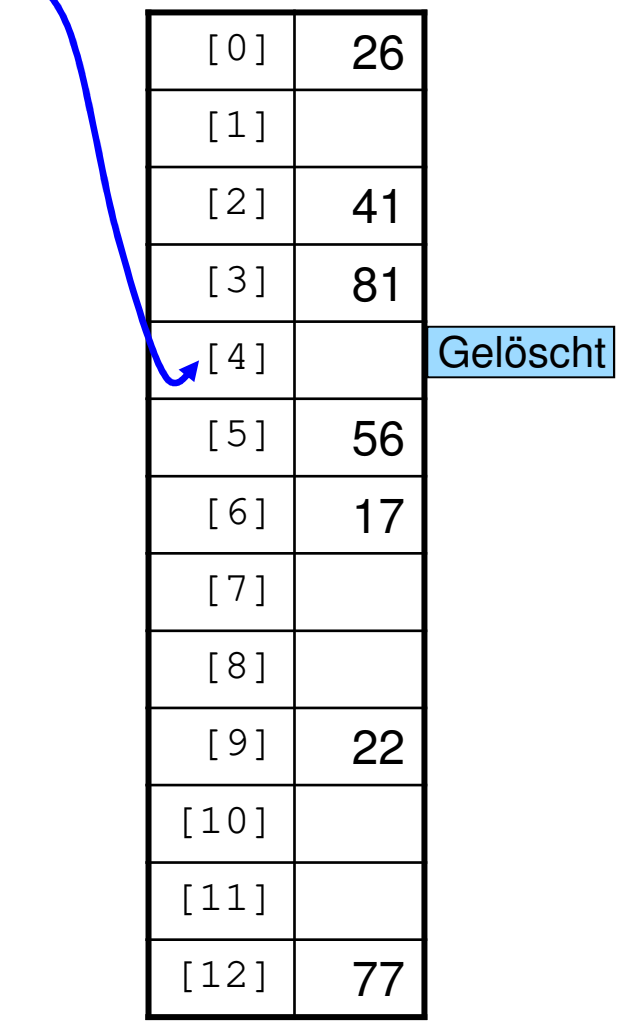
- Löschen von 69:
  - $h(69) = 69\ MOD\ 13 = 4$



Durch Kollisionen verschobene Elemente reorganisieren

Löschen (IV): Markierung

- Löschen von 69:
  - $h(69) = 69\ MOD\ 13 = 4$



Löschen (V)

- Bewertung:
  - Reorganisation:
    - Sehr aufwändig
    - Daher in Praxis kaum praktikabel
  - Markierung:
    - Performant
    - Bei häufigem Löschen: Sondierketten länger als nötig

Übersicht

- Grundbegriffe
- Kollisionsbehandlung
- Suchen und Löschen
- Bewertung

Bewertung Hashing

- Aufwand bei geringer Kollisionswahrscheinlichkeit:
  - Suchen und Einfügen in O(1)
  - Löschen:
    - O(1) bei Markierungsstrategie
    - O(n) bei Reorganisation

- Belegungsgrad Hash-Tabelle bei Sondierung:
  - Dramatische Verschlechterung von Einfüge- und Suchverhalten ab Füllungsgrad von 80%

- Nachteile gegenüber Bäumen:
  - Keine Unterstützung von Reihenfolge berücksichtigende Operationen:
    - Finde Minimum bzw. Maximum
    - Bestimme Vorgänger bzw. Nachfolger
    - Werte aus bestimmtem Bereich
    - Alle Werte geordnet ausgeben

Andere Anwendungsbereiche

- Hashing nicht nur zum Suchen
- Andere Anwendungen:
  - Kontrollsummen, z.B. bei Downloads
  - Verschlüsselung von Passwörtern
  - Signaturen
- Bekannte Algorithmen:
  - MD5 (Message-Digest Algorithm)
  - SHA-1 (Secure Hash Algorithm): 160 Bit-Hashwert
  - SHA-2:
    - SHA-256, SHA-384 und SHA-512

Zusammenfassung (I)

- Grundbegriffe:
  - Hashing
  - Hash-Funktion
  - Hash-Tabelle
  - Eigenschaften Hash-Funktion
  - Konstruktion von Hash-Funktionen:
    - Divisionsmethode
    - Folding
    - Mid-Square-Methode

- Kollisionsbehandlung:
  - Verkettete Überläufer
  - Sondierungen:
    - Lineares Sondieren
    - Quadratisches Sondieren
  - Überlaufbereich

Zusammenfassung (II)

- Suchen und Löschen
  - Suchen, Sondierungsketten
  - Löschen durch Reorganisieren
  - Löschen durch Markieren

- Bewertung:
  - Effizienz
  - Andere Anwendungsgebiete

Übungen (I)

- **Aufgabe 1**

  Finden Sie im folgenden Rätsel fünf Begriffe aus diesem Vorlesungsabschnitt!

```
H M G H Z F R K O Y I Y Q B I L Z N O L G E I M Y
M U Z E T B E W X F B G Y U M U J O V Q T V S I T
Z G B U D Z M X F B V L X C T Z V I B T V W U H T
Q N S G D D J T M S E B R F Z E V T E E U S R S H
C A J Q N S N K S X A Y F G X W U K P Q T I Y G F
K O A E W J I J B N R C S J B G R N P R M Q C S U
H K I X N G K Y S Z T J W T Q E I U K J N U Y A T
Q C Y J C A G K J L Q U L Y I S Z F H W Y O W N X
B N I O I J D H Y C G G M D L W O H V J L Y N Y V
U D S U C Y T Y I G Q W N R E M G S N U X K E A F
V A H A N R G S G M D O K G A V N A H D B C Z B Q
U F X B W W I S P C S S P U F S M H B P P H E R C
Z Y K H J A P R O M F E J G L Y N Z C D H X S H A
H D X Z E H B T K C F S M K X F T O M I G X P S C
H A S H T A B E L L E O A K L M Z R I L M H G M T
T V Z E Q P L P D D B M V K I L V Y M S F C O I R
B Z L H I C U T W U C X H Z E V M Y W E I J P Z Z
X D L X T J A R G R X T W H R P E D C W Y L M J C
R E F U E A L R E B E U E T E T T E K R E V L G O
K P F Q P I P X N K Y M I P R M Y F U N T C O O M
N Z L K O E N P Y E X H M A O F F H G P X B W H K
E I Z J H M K Q Y I N S P U G I O U U N Y E G M I
G R Q A D J O J U K S N Z I X E N G L V D B F B A
Z E H Z G D P X F M Y B Q R I G C L U R U S Z H Q
K M K L P E P L G U Q B N W X Y B Y Y G Z V J C N
```

Übungen (II)

- **Aufgabe 2**
  a) Was ist perfektes Hashing?
  b) Was ist eine Kollision?
  c) Welche Verfahren zur Kollisionsbehandlung gibt es?
  d) Welche Eigenschaften sollte eine Hash-Funktion haben?
  e) Welche Sondierungsstrategien gibt es?
  f) Was ist bei Einsatz einer Sondierungsstrategie beim Suchen zu beachten?
  g) Was ist bei Einsatz einer Sondierungsstrategie beim Löschen zu beachten?
  h) Welche Anwendungen findet Hashing neben dem Suchen?

Übungen (III)

- **Aufgabe 3**

  Gegeben sei die folgenden Zahlenfolge: 23, 24, 41, 34, 59, 61, 63, 55, 16, 84.

  a) Fügen Sie diese nacheinander in eine Hash-Tabelle mit verketteten Überläufern ein! Als Hash-Funktion soll hierbei h(x) = x MOD 9 verwendet werden!

  b) Fügen Sie die Zahlenfolge nun nacheinander in eine Hash-Tabelle als Feld ein! Es soll die gleiche Hash-Funktion verwendet werden und eine lineare Sondierungsstrategie angewendet werden!

  Übersteigt der Belegungsgrad 80% der Kapazität der Hash-Tabelle, so wird deren Kapazität um 50% erhöht. Die neue Hash-Funktion lautet dann x MOD m mit m Größe der neuen Tabelle.

- **Aufgabe 4**

  Welche zwei Strategien sind beim Löschen in Hash-Tabellen denkbar? Nennen und erläutern Sie diese kurz und nennen Sie jeweils ein Problem, das bei den Strategien auftritt!

Übungen (IV)

- **Aufgabe 5**

	wahr	falsch
Eine Hash-Funktion h(x) = x mod n arbeitet am besten, wenn n eine Primzahl ist.		
Ab einem Füllungsgrad von 30% ist das Einfügen in Hash-Tabellen nicht mehr effizient.		
Kollisionen können beim Hashing nur entstehen, wenn zweimal der gleiche Wert eingefügt werden soll.		
Hashing ist gut geeignet, wenn relativ selten gelöscht wird.		
Hash-Verfahren sind zum schnellen Suchen stets Baumstrukturen vorzuziehen.		
Bei geringer Kollisionswahrscheinlichkeit hat das Einfügen in eine Hash-Tabelle einen Aufwand $O(\log_2 n)$.		
Quadratisches Sondieren erleichtert gegenüber linearem Sondieren das Löschen von Einträgen.		