Schnelles Suchen und Hashing

- Sie kennen die Begriffe: Vor-/Nachbedingung und Invariante
- Sie wissen, wie schnell gesucht werden kann
- Sie wissen, wie binäres Suchen funktioniert
- Sie wissen, wie Hashing funktioniert
- Sie kennen die Java Datenstruktur Map und HashMap

Basiert auf Material von:

Kurt Bleisch
Stephan Neuhaus
Karl Rege
Marcela Ruiz
Jürgen Spielberger

Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

Suchen

Suchen: Beispiele

- Prüfen ob ein Wort in einem Text vorkommt
- Zählen wie oft ein Wort in einem Text vorkommt
- Überprüfen ob alle Worte korrekt geschrieben sind
- Finden einer Telefonnummer in einer Telefonbuch
- Prüfen ob die richtige Zahlenkombination im Lotto gewählt wurde
- Prüfen ob eine Kreditkartennummer gesperrt ist
- Prüfen ob in zwei Listen die gleichen Elemente vorkommen
- Usw.

Suchen: Vor-, Nachbedingung und Invariante

Vorbedingung: Aussage, die vor dem Ausführen der Programmsequenz gilt.

Nachbedingung: Aussage, die nach dem Ausführen der Programmsequenz gilt.

{x >= 0} y = sqrt(x) {y = √x, x >= 0}

Vorbedingung

Nachbedingung

Invariante: Aussage, die über die Ausführung hinweg gültig bleibt.

```
k = 0;
```

$\{\forall x_i; i < k; x_i \neq S\} \wedge k = 0$

```
while (x[k] != S && k < x.length) k++;
```

$\{\forall x_i; i < k; x_i \neq S\} \wedge \{x_k = S \vee k >= x.length\}$

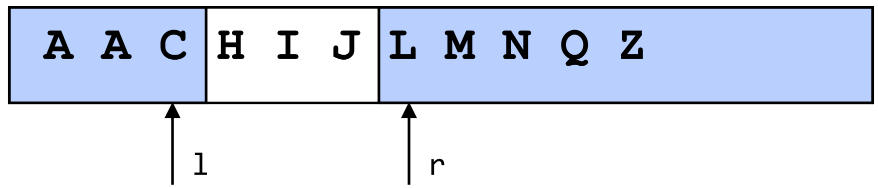
Schleifeninvariante

Bereich

Aussage

Verneinung der Schleifenbedingung.

Suchen: Binäres Suchen im sortierten Array



Gegeben sei ein **sortiertes** Array von Werten (Buchstaben). Wie kann ein Wert S in so einem Array effizient gesucht werden?

- Führe zwei Indizes ein: `l` und `r`
- Invariante:

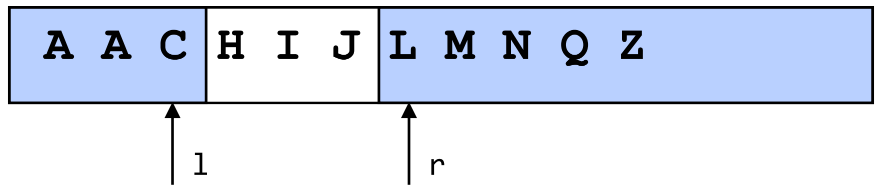
$$\forall k, n; k \leq l, n \geq r; a[k] < S \wedge a[n] > S$$

So lange die Invariante gilt, so lange haben wir S nicht gefunden, liegt S «zwischen» l und r.

Verwendung:

1. Invarianten können zum Beweis der Korrektheit von Algorithmen verwendet werden.
2. Verwendung im Design By Contract: Für eine Methode werden alle Vor- und Nachbedingungen und Invarianten in ihrem Ablauf beschrieben.

Suchen: Binäres Suchen im sortierten Array



Wiederhole bis S gefunden oder nicht gefunden:

1. nehme `m` als Index zwischen `l` und `r`
2. falls `a[m] < S` → `l = m`
3. falls `a[m] > S` → `r = m`
4. falls `a[m] == S` → gefunden
5. falls `l + 1 >= r` → nicht gefunden, keine Elemente mehr zwischen l und r

$$\forall k, n; k \leq l, n \geq r; a[k] < S \land a[n] > S$$

Invariante: Bleibt gültig, auch wenn wir S gefunden haben.

Suchen: Binäres Suchen im sortierten Array

In jedem Durchgang wird `r - l` halbiert → $\log_2$ Schritte

```
static int binary(int[] a, int s) {
   int l = -1;
   int r = a.length;
   int m = (l + r) / 2;
   // Invariante && l == -1 && r == a.length
   while (l != r && a[m] != s) {
      if (a[m] < s) l = m;
      else r = m;
      m = (l + r) / 2;
   }
   // Invariante && (l == r || a[m] == s)
   return (a[m] == s)?m:-1;
}
```

l und r nach der Initialisierung ausserhalb der Arraygrenzen, daher ist die Invariante auch zu Beginn gültig.

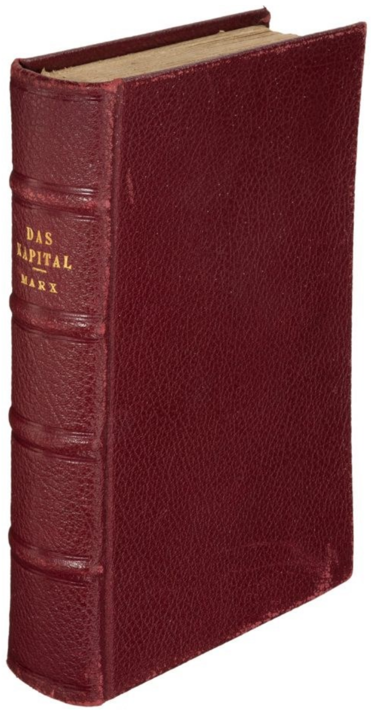
Aufwand: $O(\log_2 n)$ – aber?

Das Finden ist mit wenigen Schritten getan. Das Array muss aber sortiert sein (kostet etwas), sonst funktioniert das nicht!

$$\forall k, n; k \leq l, n \geq r; a[k] < S \wedge a[n] > S$$

Invariante

Suchen: In mehreren Listen



Fragen:

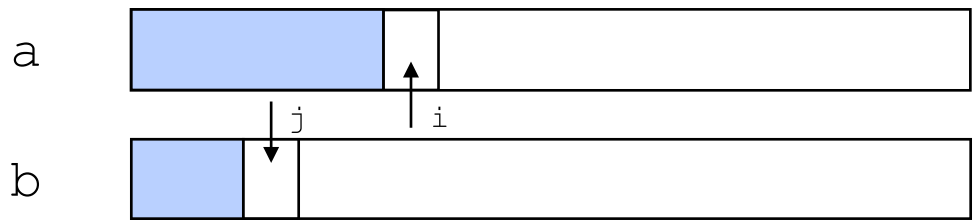
- Welche der 300 Reichsten der Schweiz beziehen Krankenkassenvergünstigung?
- Welcher Student, dessen Eltern reich sind, bekommt ein Stipendium?
- Welcher LinkedIn-User hat «Das Kapital» von Karl Marx gekauft?

Suchen: In mehreren Listen
Einfacher Algorithmus

```
static int indexOf(String[] a, String[] b) {
   for (int i = 0; i < a.length; i++) {
      for (int j = 0; j < b.length; j++) {
         if(a[i].equals(b[j])) return i;
      }
   }
   return -1;
}
```

- Doppelt geschachtelte Schleife.
- Aufwand O(n·m), bei 2 · 10'000 Elementen → 10^8

Suchen: In mehreren Listen
Besserer Algorithmus wenn a und b sortiert



$$\forall k, n;\ k < j, n < i;\ b[k] \neq a[n]$$

Invariante

Bereich, für den die Invariante gilt, sukzessive erweitern:

`b[j] < a[i]` → $\forall k; k <= j; b[k] < a[i]$ → `j` um 1 erhöhen

`b[j] > a[i]` → $\forall n; n <= i; b[j] > a[n]$ → `i` um 1 erhöhen

`b[j] = a[i]` → gefunden Wert zurückgeben.

Suchen: In mehreren Listen

```
static int indexOf(String[] a, String[] b) {
   int i = 0, j = 0;
   // Invariante && i == 0 && j == 0
   while (!a[i].equals(b[j]) && (i < a.length-1 || j < b.length-1)) {
      int c = a[i].compareTo(b[j]);
      if (c < 0 || j == b.length-1 ) i++;
      else if (c > 0 || i == a.length-1) j++;
   }
   // Invariante && (i == a.length-1 && j == b.length-1) || (a[i] == b[j])
   if (a[i].equals(b[j])) return i; else return -1;
}
```

Schleife wird verlassen wenn diese Bedingung nicht mehr gilt → Negation der Bedingung gilt am Schluss.

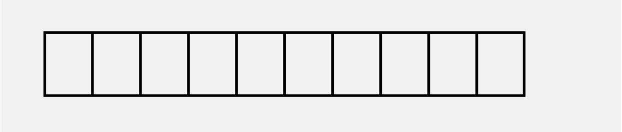
$$\forall k, n;\ k < j, n < i;\ b[k] \neq a[n]$$

Invariante

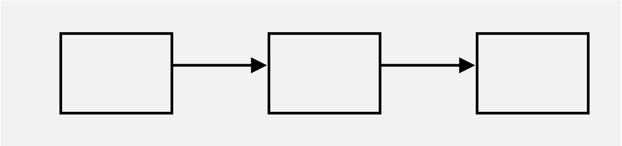
- i und j werden erhöht, so dass die Invariante erhalten bleibt.
- Am Schluss gilt: Invariante & Abbruchbedingung der Schleife.
- Aufwand: O(n) – aber?

Suchen: Aufwand für Suchen und Einfügen

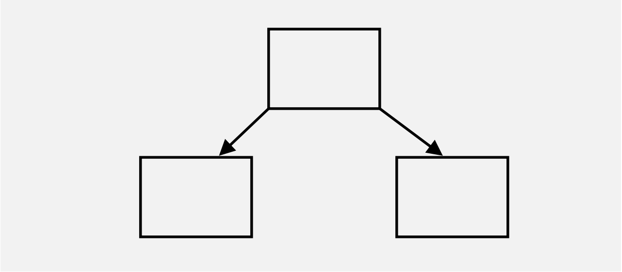
- **Sortierter Array**
  - Einfügen: O(n/2) → O(n)
  - Binäres Suchen: $O(\log_2(n)) \rightarrow O(\log(n))$



- **Lineare sortierte Liste**
  - Einfügen: O(n/2) → O(n)
  - Suchen: O(n/2) → O(n)



- **Sortierter Binärbaum**
  - Einfügen: $O(\log_2(n)) \rightarrow O(\log(n))$
  - Suchen: $O(\log_2(n)) \rightarrow O(\log(n))$



Frage:

Gibt es ein «Such-Verfahren», dessen Aufwand unabhängig von der Anz. Elemente ist?

Hashing

Hashing: Schlüssel → Inhalt

Gegeben sei eine Menge von Datensätzen der Form:

Schlüssel	Inhalt

- Der Schlüssel kann ein Teil des Inhalts sein.
- Der Schlüssel besteht im einfachsten Fall aus einem String oder einem numerischen Wert.
- Mittels des Schlüssels kann der Datensatz wiedergefunden werden. Bsp:
  - AHV-Nummer → Personen
  - Matrikel-Nummer → Studenten
- Aufgabe: Es sollen Daten (Objekte) in einen Behälter eingefügt und mittels ihres Schlüssels wiedergefunden werden.

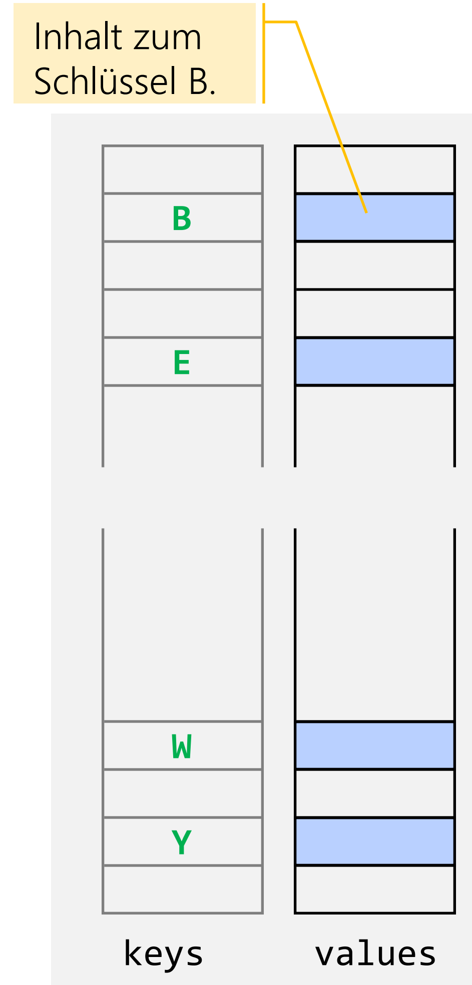
Hashing: Idee

Werte in Array an ihrer Indexposition (z.B. bestimmt durch ASCII-Code) speichern.

- Wertebereich: Alle Schlüsselwerte A…Z
- Schlüsselwerte im Beispiel: B, E, W, Y
- Array: `char[] values = new char[256]`

- Einfügen: `values[key] = value`
- Suchen: `value = values[key]`

- Aufwand:
  - Einfügen O(1)
  - Suchen O(1)



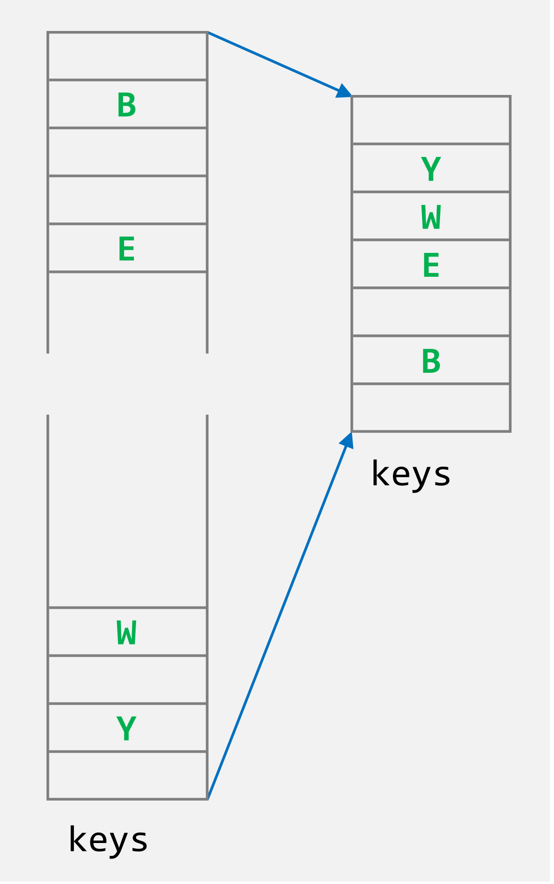
Hashing: Idee

- Probleme:
  - Der Array ist nur schwach belegt.
  - Geht noch für Buchstaben, was aber bei Zahlen ($>2^{32}$) oder Strings?
- Lösung:
  - Es wird eine Funktion verwendet, welche den grossen Wertebereich auf einen kleineren Bereich abbildet.
  - Einfache Funktion: X mod tableSize → Zahl zwischen 0 und tableSize-1
  - Eine solche Funktion nennt man Hash-Funktion.

- Wirklich gute Hashfunktionen sind nicht immer einfach zu finden (gleichmässige Verteilung), zwei gebräuchliche Hashfunktionen:
  1. $h(k) = k \bmod M$
     M ist häufig Primzahl
  2. $h(k) = \lfloor m \cdot ((k \cdot c) \bmod 1) \rfloor$
     m = Anz. Hashadressen, $0 < c < 1$; c = z.B. goldener Schnitt (0,618033…)

Divisonsrest-Methode

Multiplikations-Methode



Hashing: Algorithmus, Kollision vernachlässigt

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering

```
public class Hashtable {
   final int MAX = 97;
   final int INVALID = Integer.MINVALUE;
   int[] keys = new int[MAX];
   int[] vals = new int[MAX];

   private void init()
   {
      for (int i = 0; i < MAX; i++) {
         key[i] = INVALID;
      }
   }

   private int h(int key) {
      return key % MAX;
   }
```

Hash-Funktion

```
   public void put(int key, int val) {
      int h = h(key);
      if (keys[h] == INVALID) {
         keys[h] = key;
         vals[h] = val;
      }
      else {/* COLLISION */}
   }

   public int get(int key) {
      int h = h(key);
      if (keys[h] == key) {
         return vals[h];
      }
      else return INVALID;
   }
}
```

Überprüfe ob Feld frei.

Speichere Wert.

Schlüssel vorhanden?

Hole Wert.

Hashing: Probleme

Zürcher Hochschule für Angewandte Wissenschaften
zhaw School of Engineering

Problem 1:

- 💣 Aus dem Hash-Wert kann der ursprüngliche Wert nicht mehr bestimmt werden.

Lösung:

- ⇒ Originalwert ebenfalls in Tabelle (z.B. zusätzlicher Array) speichern.

Problem 2:

- 💣 Suche einer geeigneten Hash-Funktion.

Lösung:

- ⇒ Schwierig, Kenntnisse der zu erwarteten Schlüssel sinnvoll.

Problem 3:

- 💣 Zwei unterschiedliche Objekte können den gleichen Hash-Wert haben, d.h. sie müssten an der gleichen Stelle (Massierungen / Clustering) gespeichert werden ⇒ Kollision.

Lösung:

- ⇒ Kollisionen werden vermieden (Bildbereich gleich gross wie Ur-Bildbereich) - oder verringert (suche einer geeigneten Hash-Funktion).
- ⇒ Kollision wird aufgelöst, verschiedene Verfahren zur Auflösung (später): linear / quadratic probing, …

Hashing: Problem Hash-Funktion

*1	*256	*256^2	*256^3
B	A	U	M

- Hashfunktion: ASCII-Wert × $256^{\text{Position-1}}$
- Es entstehen sehr grosse Zahlen

  Ein vier Zeichen langer ASCII-String führt bereits zu einer Zahl in der Grössenordnung von $256^4 = 2^{32}$.
  Falls das Zeichen 32-Bit verwendet, dann sogar $4'294'967'296^4 = 2^{128} = 3.4 \cdot 10^{38}$.
- In der Praxis wird zur Umwandlung von Strings das **Horner-Schema** verwendet:

  $A_3x^3 + A_2x^2 + A_1x^1 + A_0x^0 = (((A_3)x + A_2)x + A_1)x + A_0$, trotzdem bleibt das Overflow-Problem.
- Mit dem Modulo-Operator wird die Menge der verschiedenen Hashwerte eingeschränkt: $(A_3x^3 + A_2x^2 + A_1x^1 + A_0x^0) \bmod n$

  Da $(a + b) \bmod n = (a \bmod n + b \bmod n) \bmod n$ → $((A_3x^3) \bmod n + (A_2x^2) \bmod n + (A_1x^1) \bmod n + (A_0x^0) \bmod n) \bmod n$. Trotzdem bleibt Problem mit dem grössten Term.

Hashing: hashCode()-Methode in Java

Hashwerte und equals müssen folgenden «Vertrag» (Contract) einhalten

1. Ein Objekt muss während seiner **Lebensdauer immer denselben Hashwert** zurückliefern, solange der Zustand (Wert) des Objekts nicht verändert wurde.
   Wenn aber die JVM neu gestartet oder eine andere JVM verwendet wird, dann darf der Hashwert ändern!
2. wenn **equals() == true**, dann **müssen** die Objekte **denselben Hashwert** liefern
3. wenn **equals() == false**, dann **sollten** die Objekte **unterschiedliche Hashwerte** liefern (d.h. Haswerte müssen nicht eindeutig sein).

Zusätzlich:

- wenn **equals() == true**, dann **muss** compareTo() == 0 liefern
- wenn **equals() == false**, dann **muss** compareTo() != 0 liefern

Immer equals, compareTo und hashCode zusammen überschreiben.

Hashing: hashCode-Implementierung in Java

Bei Klassen kann der Hashcode z.B. aus den Werten der Felder berechnet werden.

```
public class Employee {
   int employeeId;
   String name;
   Department dept;

   @Override
   public int hashCode() {
      int hash = 1;
      hash = hash * 13 + employeeId;
      hash = hash * 17 + name.hashCode();
      hash = hash * 31 + (dept == null ? 0 : dept.hashCode());
      return hash;
   }
}
```

Ohne Modulo, da noch unabhängig von der Anwendung (d.h. Grösse) der Hashtabelle.

Primzahlen

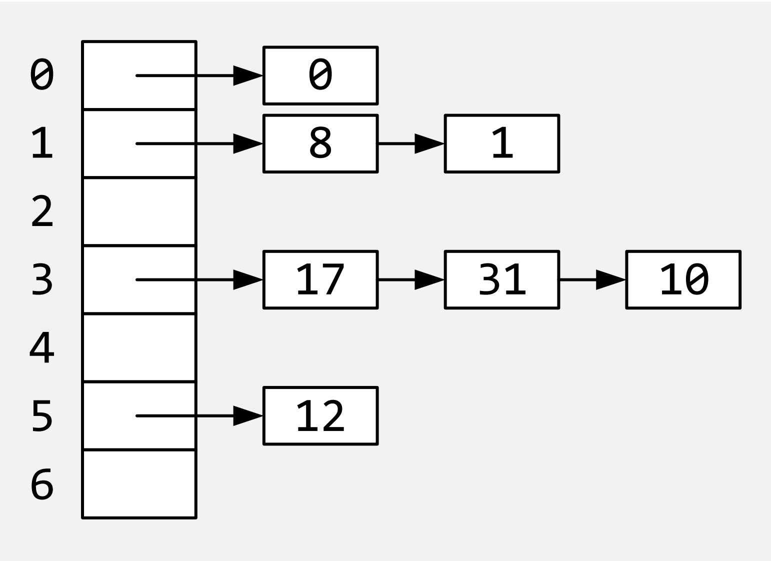
Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

Kollisionen

Kollisionsauflösung 1: Überlauflisten

Hashtable lediglich als Ankerpunkt für Listen aller Objekte, die den gleichen HashWert haben: **Überlauflisten** (Separate Chaining).



- Nachteil: Overhead durch Verwendung einer weiteren Datenstruktur.
- Vorteil: Funktioniert auch noch bei Load-Faktor λ nahe oder gleich 1.

Anzahl Einträge / Anzahl Plätze

Kollisionsauflösung 1: Übung

Gegeben sind:

- Eine Hashtabelle der Grösse 10.
- Eine Hash-Funktion h(x) = x mod 10.
- Input: 4371, 1323, 6173, 4199, 4344, 9679, 1989.

Wie sieht die Tabelle aus, nachdem der Input unter Verwendung von Separate Chaining Hashing (Überlaufliste) verarbeitet wurde?

0
1 4371
2
3 1323 6173
4 4344
5
6
7
8
9 4199, 9679, 1989

Kollisionsauflösung 2: Open Addressing

Open Addressing: Techniken, bei welchen bei einer Kollision eine freie Zelle sonst wo in der HashTable gesucht wird ⇒ setzt einen Load-Faktor < ~0.8 voraus.

Lineares Sondieren (Linear Probing):

Sequentiell nach nächster freier Zelle F+1, F+2, F+3, ..., F+i suchen (mit Wrap around).

Quadratisches Sondieren (Quadratic Probing):

In wachsenden Schritten der Reihe nach F+1, F+4, F+9, . . . , F+i^2 prüfen (mit Wrap around).

Kollisionsauflösung 2: Lineares Sondieren

In eine Hash-Tabelle mit 10 Feldern werden der Reihe nach 89, 18, 49, 58 und 9 eingefügt.

0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

- Hash-Funktion: Input modulo Tabellengrösse.
- Bei zunehmendem Load-Faktor dauert es immer länger bis eine Zelle gefunden wird (Einfügen und Suchen).
- `find` funktioniert wie `insert`: Element wird in Tabelle ausgehend vom Hash-Wert gesucht bis Wert oder leere Zelle gefunden wird.

Kollisionsauflösung 2: Lineares Sondieren

- **Performance**:
  Ziemlich schwierig zu bestimmen, da der Aufwand nicht nur vom Load-Faktor, sondern auch von der Verteilung der belegten Zellen abhängt, aber i.d.R. O(1).

- Phänomen des **Primary-Clustering:**
  Mussten einmal freie Zellen neben dem Hash-Wert belegt werden, steigt die Wahrscheinlichkeit, dass weiter gesucht werden muss für:
  - alle Ausgangswerte mit gleichem Hash-Wert
  - all jene, deren Hash-Wert in eine der nachfolgenden Zellen verweist.

  Folge:
  - Verlängerung des durchschnittlichen Zeitaufwandes zum Sondieren.
  - Erhöhte Wahrscheinlichkeit, dass weiteres Sondieren nötig wird.

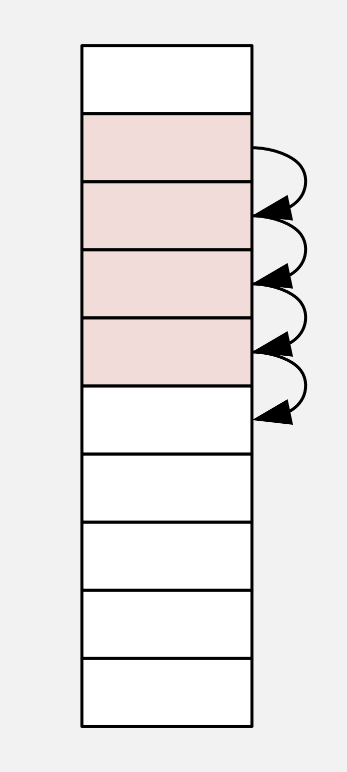
$\Rightarrow$ Bei hohem Load-Faktor oder ungünstigen Daten bricht die Performance ein!

Kollisionsauflösung 2: Lineares Sondieren

```
int find(Object x ) {
    int currentPos = hash(x);

    while (array[ currentPos ] != null &&
       !array[currentPos].element.equals( x ) ) {
       currentPos = (currentPos  + 1) % array.length;
    }
    return currentPos;
}
```

Die Suche muss so lange fortgesetzt werden, bis das Element gefunden wurde oder die Zelle leer ist.



Kollisionsauflösung 2: Übung

Gegeben sind:

- Eine Hashtabelle der Grösse 10
- Eine Hash-Funktion h(x) = x mod 10
- Input: 4371, 1323, 6173, 4199, 4344, 9679, 1989.

Wie sieht die Tabelle aus, nachdem der Input unter Verwendung einer linearen Sondiermethode verarbeitet wurde?

Kollisionsauflösung 2: Quadratisches Sondieren

In eine Hash-Tabelle mit 10 Feldern werden der Reihe nach 89, 18, 49, 58 und 9 eingefügt.

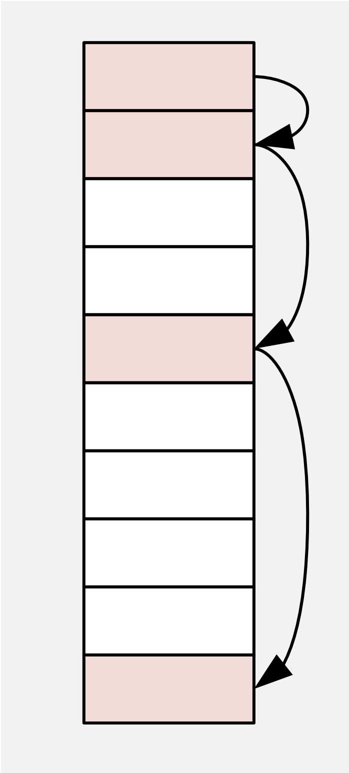
0			49	49	49
1					
2				58	58
3					9
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

- Hash-Funktion: Input modulo Tabellengrösse.
- Jetzt bleiben Lücken in der Hash-Tabelle offen.
- Die zuletzt eingefügte 9 findet ihren Platz unbeeinflusst von der zuvor eingefügten 58.

Kollisionsauflösung 2: Quadratisches Sondieren

```
int findPos( Object x ) {
    int collisionNum = 0;
    int currentPos = hash(x);

    while (array[currentPos] != null &&
       !array[currentPos].element.equals(x))
    {
       currentPos += 2 * ++collisionNum - 1;
       currentPos = currentPos % array.length;
    }
    return currentPos;
}
```



Bessere Performance als lineares Probing weil weniger Primary-Clustering auftritt.

Kollisionsauflösung 2: Übung

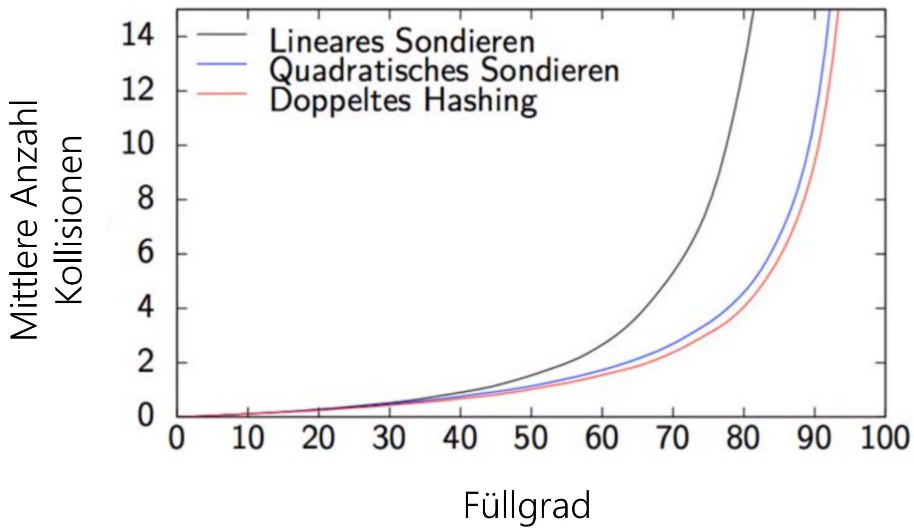
Gegeben sind:

- Eine Hashtabelle der Grösse 10
- Eine Hash-Funktion h(x) = x mod 10
- Input: 4371, 1323, 6173, 4199, 4344, 9679, 1989.

Wie sieht die Tabelle aus, nachdem der Input unter Verwendung einer quadr. Sondiermethode verarbeitet wurde?

Kollisionen: Load-Faktor

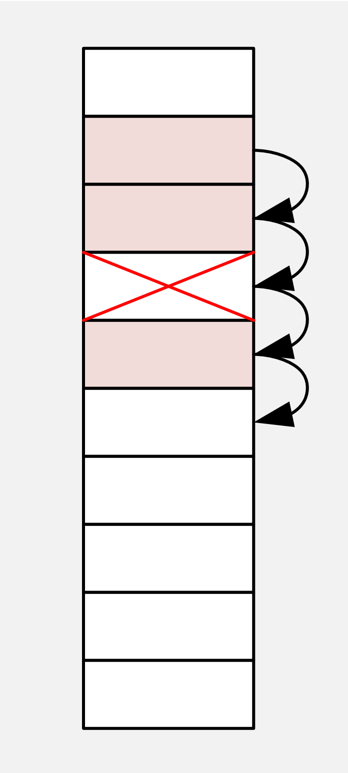
- Anzahl Kollisionen hängt von der Güte der Hash-Funktion und der Belegung der Zellen ab.
- Der Load-Faktor λ (Anzahl Einträge / Anzahl Plätze):
  - Sagt wie stark der Hash-Bereich belegt ist.
  - Bewegt sich zwischen 0 und 1.
  - Die Anzahl Kollisionen ist abhängig vom Load-Faktor λ und der Hash-Funktion h: $f(h, \lambda)$



Joost-Pieter Katoen RWTH Aachen

Kollisionsauflösung: Löschen

- Werte können nicht einfach gelöscht werden, da sie die Folge der Ausweichzellen unterbrechen.
- Wenn ein Wert gelöscht wird, müssen alle Werte, die potentielle Ausweichzellen sind, gelöscht und wieder eingefügt werden (rehashing).
- Zweite Möglichkeit: gelöschte Zelle lediglich als «gelöscht» markieren.



Hashing: Vor- und Nachteile

Vorteile:

- Suchen Einfügen in Hash-Tabellen sehr effizient.
- «Einfache» binäre Bäume können bei ungünstigen Inputdaten degenerieren, Hash-Tabellen kaum.
- Der Implementationsaufwand für Hash-Tabellen ist geringer als derjenige für ausgeglichene, binäre Bäume.

Nachteile:

- Das kleinste oder grösste Element lässt sich nicht einfach finden.
- Geordnete Ausgabe ist nicht möglich.
- Die Suche nach Werten in einem bestimmten Bereich oder das Finden z.B. eines Strings, wenn nur der Anfang bekannt ist, ist nicht möglich.

Hash-Tabellen sind geeignet wenn: die **Reihenfolge** nicht von Bedeutung ist, nicht nach **Bereichen** gesucht werden muss und die **ungefähre (maximale) Anzahl** bekannt ist.

Extendible Hashing

Extendible Hashing: Auslöser

Was tun, wenn die Hashtabelle überläuft oder sogar nicht mehr in den Hauptspeicher passt?

Hashtabelle passt noch in den Hauptspeicher:

- Überlaufketten → Performance beim Zugriff wird schlechter.
- In neue, genügend grosse Hashtabelle umkopieren (rehashing mit neuer Hashfunktion) → relativ teure Operation.

Hashtabelle passt nicht mehr in den Hauptspeicher:

- Extendible Hashings:
  - Der Schlüsselwertebereich kann nachträglich vergrössert werden.
  - Funktioniert mit Files/Blockstruktur (wie B-Bäume).

Extendible Hashing: Grundidee

- Naive Implementation einer grossen Hashtabelle als File festgelegter Grösse
  - Rehashing würde bedeuten, dass sämtliche Schlüssel bzw. das ganze File neu geschrieben/umorganisiert werden muss.

- Idee: verwende **Hash-Verzeichnis** von Verweisen zu «Buckets» (= Behälter)
  - Ein Bucket enthält mehrere Einträge.
  - Die Buckets müssen nicht mehr hintereinander auf der Disk liegen.
  - Grosse Buckets (z.B. Bucket-Size = Disk-Block) haben kleine Schlüssel im Verzeichnis zur Folge.

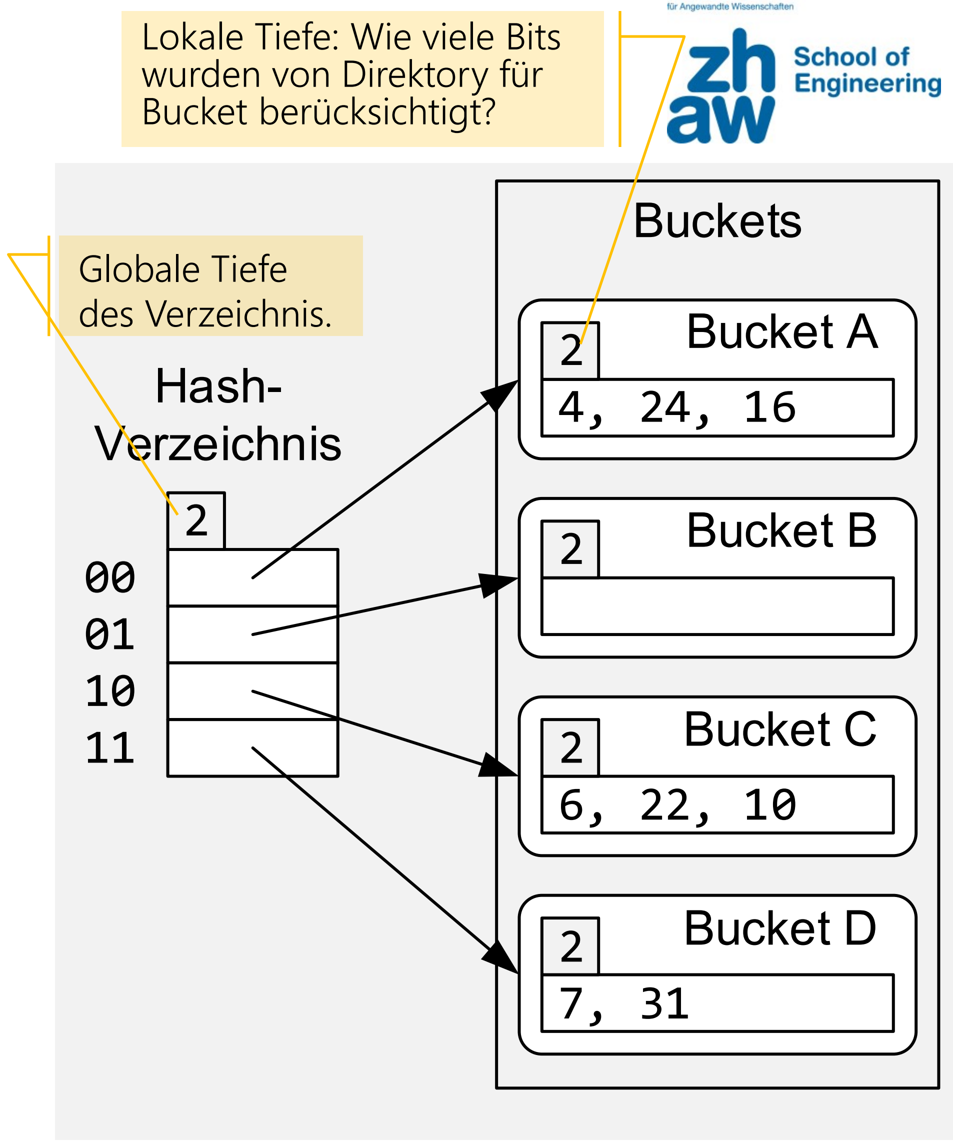
- Hash-Verzeichnis enthält lediglich (die letzten) n-Bits des Schlüssels und ist wesentlich kleiner (als die Daten selbst)
  - Das Verzeichnis kann in den Hauptspeicher geladen werden.
  - Es kann einfacher verdoppelt werden.

- Hashfunktion wird entsprechend der berücksichtigten Bits im Verzeichnis angepasst.

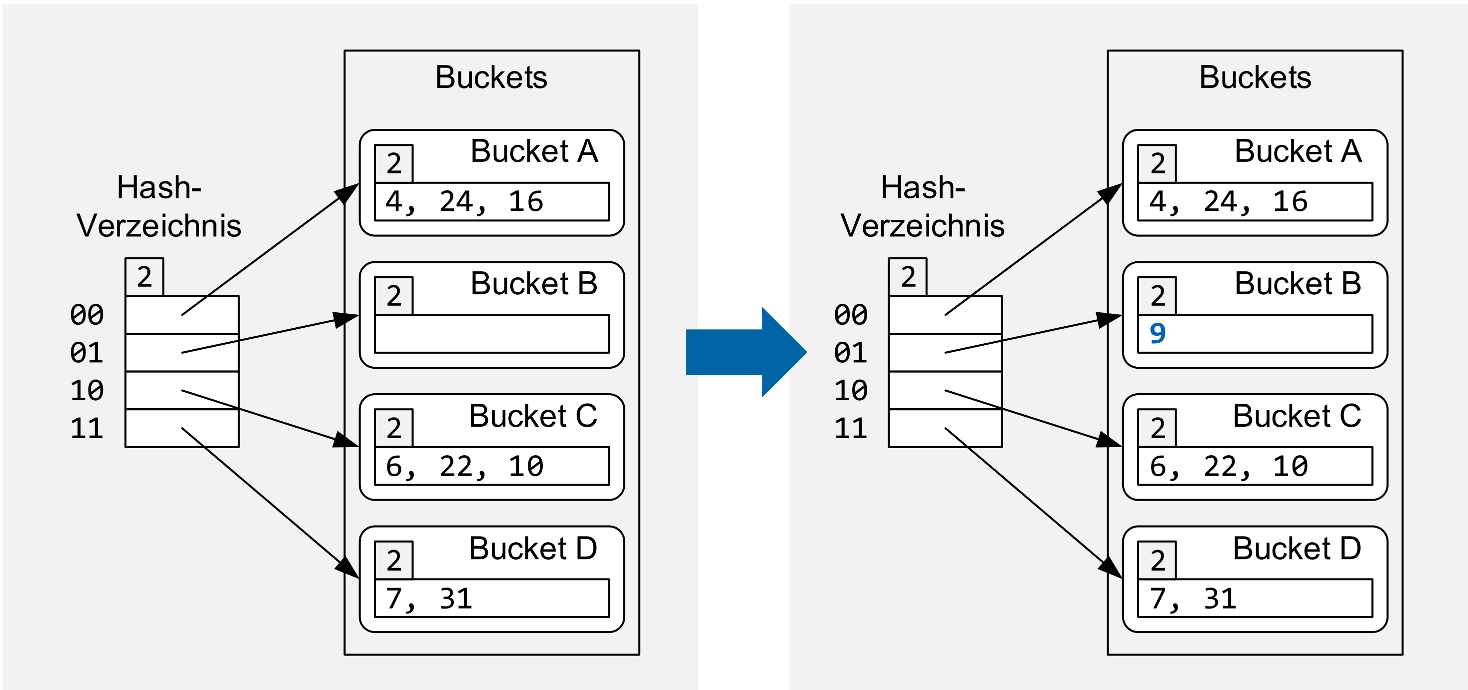
Extendible Hashing: Beispiel

- Hash-Verzeichnis ist ein Array der Grösse 4.
- Um einen Bucket zu finden, verwende die Anzahl Bits des Schlüssels gemäss globaler Tiefe.

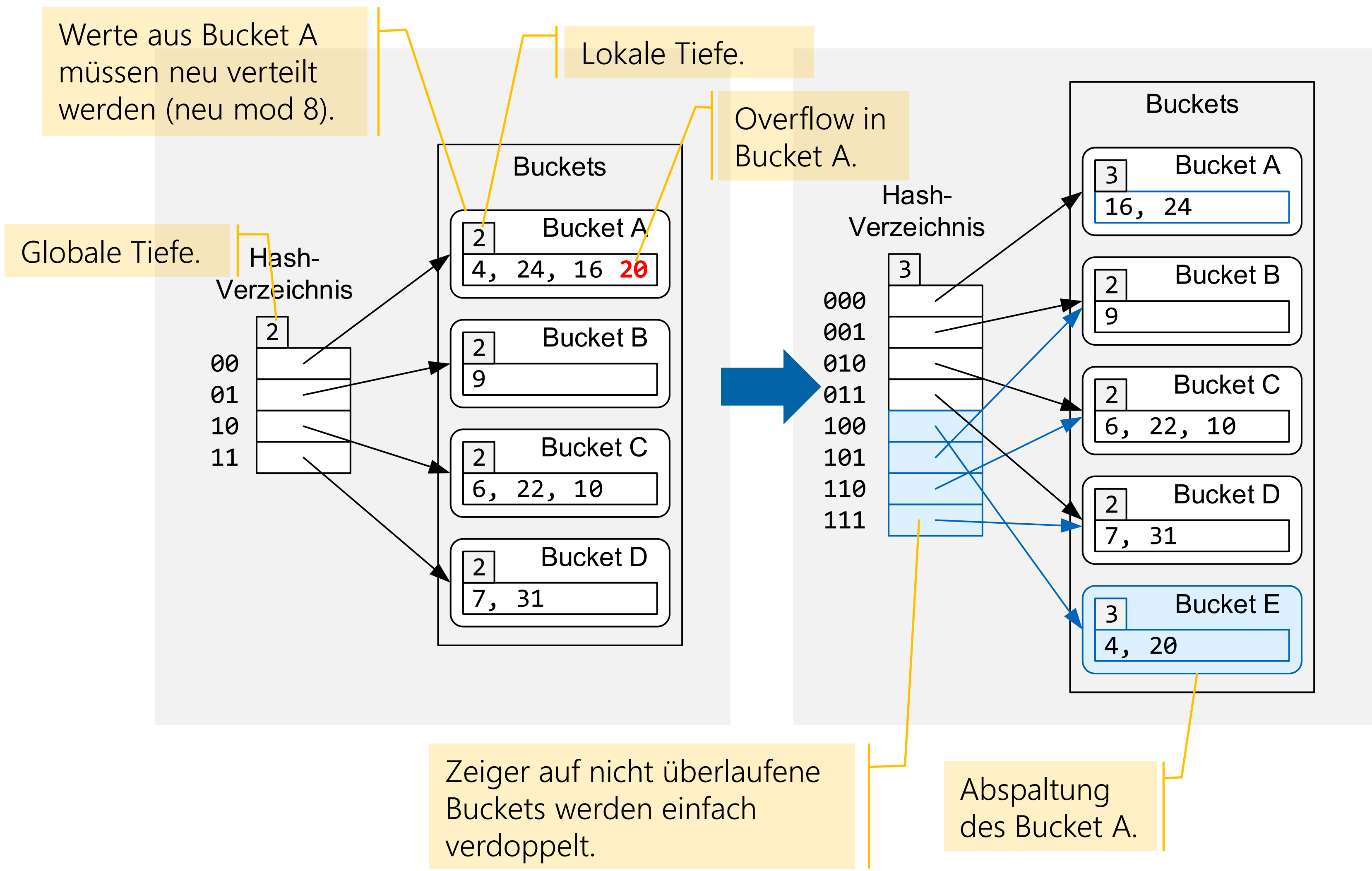
  Bsp.: Gesuchter Wert = 6 (binär: 110); dann ist Zeiger auf Bucket im Ver-zeichniseintrag 6 % 4 = 2 (binär: 10).
- **Insert**: Falls ein Bucket voll ist, dann Bucket teilen. Falls not-wendig (globale Tiefe = lokale Tiefe), dann auch das Verzeichnis verdoppeln (siehe nächste Folien).



Extendible Hashing: Einfügen ohne Overflow



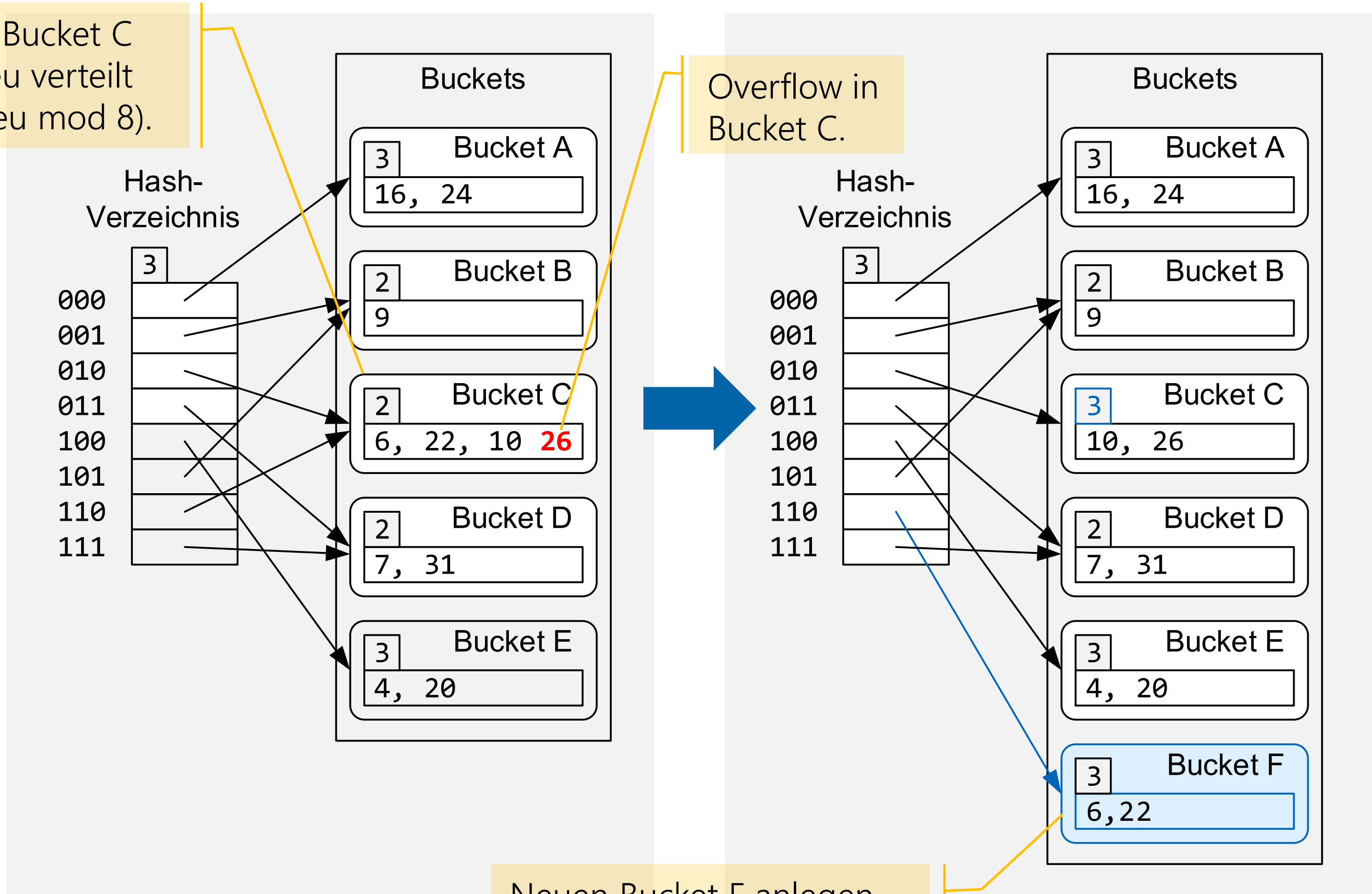
Extendible Hashing: Einfügen mit Overflow mit (globale Tiefe = lokale Tiefe)



Extendible Hashing: Einfügen mit Overflow mit (globale Tiefe > lokale Tiefe)

Werte aus Bucket C müssen neu verteilt werden (neu mod 8).

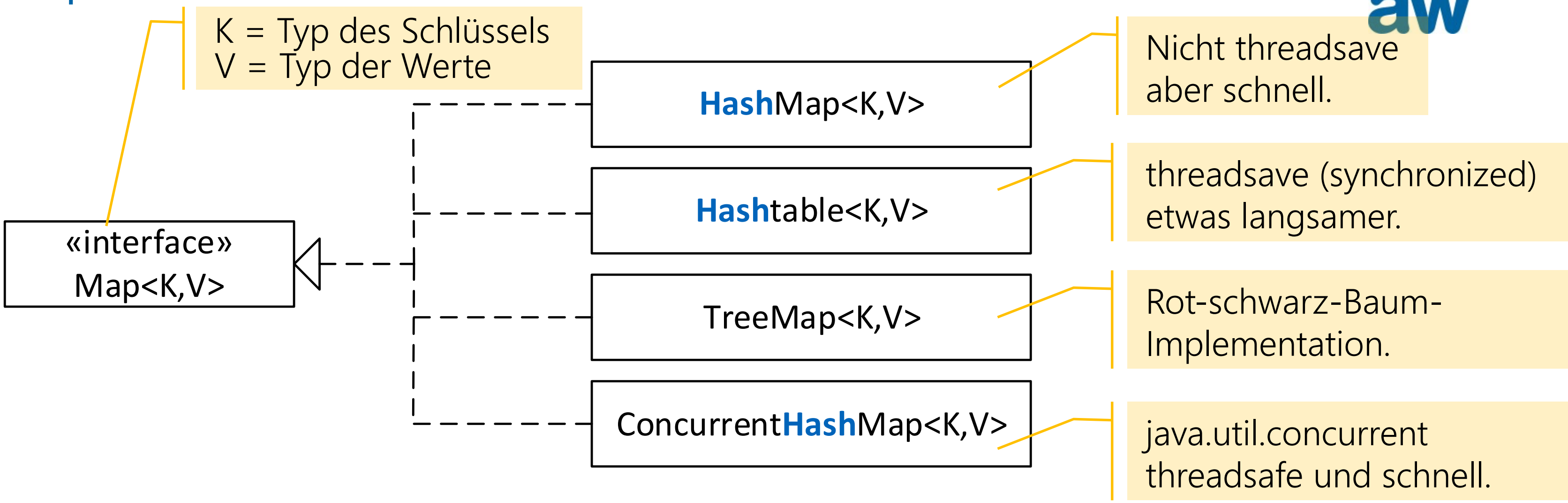
Overflow in Bucket C.



Neuen Bucket F anlegen, Pointer anpassen und Werte neu verteilen.

Implementationen in Java

Maps



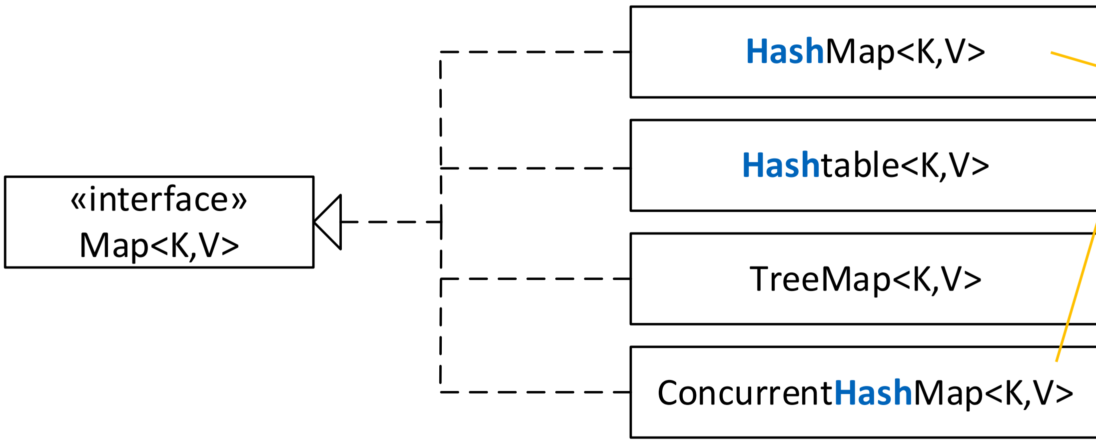
Die Default-Implementation der hashCode()-Methode liefert:

- Verschiedene Werte bei neuerlicher Codeausführung (vielleicht).
- Garantiert keine sinnvolle Verteilung der Werte.

→ Methode beim Einsatz von HashMap, etc. allenfalls geeignet überschreiben.

```
Object {
    ...
     public int hashCode();
    ...
}
```

Maps



Für neue Programme entweder HashMap oder ConcurrentHashMap verwenden.

Konstruktoren:

Initialgrösse 16, automatisch Rehashing falls Load-Faktor > 0.75.

```
HashMap<K,V>()
HashMap<K,V>(int initialCapacity)
```

Erzeugen von HashMap

Erzeugen von HashMap mit Grösse

Sinnvoll, falls die Grösse etwa bekannt ist.

Map<K, V> Interface

`void clear()`	Löschen aller Elemente
`int size()`	Anzahl Elemente
`V put(K key, V value)`	Einfügen eines Elementes
`V get(Object key)`	Finden eines Elementes
`V remove(Object key)`	Löschen eines Elementes
`boolean containsKey(Object key)`	ist Element mit Schlüssel in Tabelle
`boolean containsValue(Object value)`	hat ein Element den Wert
`Collection<V> values()`	alle Werte
`Set<K> keySet()`	alle Schlüssel als Set

```
for (String elem : h.keySet())
  System.out.println(elem);
```

Zusammenfassung

- Suche
  - Einfache Suche
  - Invariante
  - Binäre Suchen
  - Suche in zwei Sammlungen

- Hashing
  - Idee
  - Hashfunktion
    - Gute Hashfunktionen
  - Kollisionsauflösung
    - Überlauflisten
    - Lineares Sondieren
    - Quadratisches Sondieren
  - Vor- und Nachteile
  - Extended Hashing
  - Implementationen in Java

Kontrollfragen Lektion 9
nicht vergessen – heute mit
Der Wolf und die 7 Geisslein