

# Schnelles Suchen und Hashing



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

Suchen

# Beispiele wo gesucht werden muss

- 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

## Vorbedingung

- Aussage, die vor dem Ausführen der Programmsequenz gilt

## Nachbedingung

- Aussage, die nach dem Ausführen der Programmsequenz gilt

- $\{x \geq 0\} \ y = \text{sqrt}(x) \ \{y = \sqrt{x}, x \geq 0\}$

Vorbedingung

Nachbedingung

## Invariante

Invariante

Bereich

Aussage

$k = 0;$

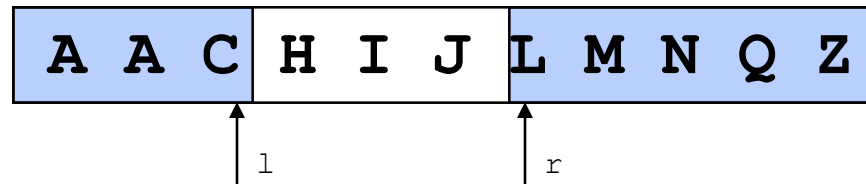
$\{\forall x_i; i < k; x_i \neq S\} \wedge k = 0$

while  $(x[k] \neq S \ \&\& \ k < x.length)$   $k++;$

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

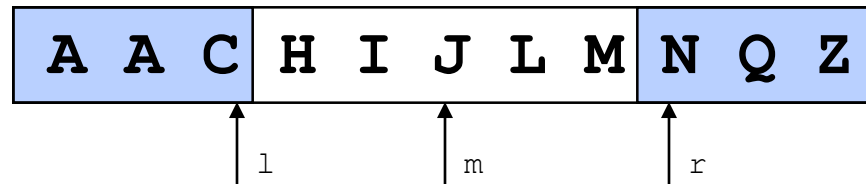
Verneinung der Schleifenbedingung

# Binäres Suchen in einem 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:  $\forall k, n; k \leq l, n \leq r; a[k] < S \wedge a[n] > S$

## ... Binäres Suchen in einem sortierten Array



- nehme  $m$  als Index zwischen  $l$  und  $r$
- falls  $a[m] < S \rightarrow l = m$
- falls  $a[m] > S \rightarrow r = m$
- falls  $a[m] == S \rightarrow$  gefunden
- falls  $l+1 \geq r \rightarrow$  keine Elemente mehr zwischen  $l$  und  $r \rightarrow$  nicht gefunden

# ... Binäres Suchen in einem sortierten Array

```
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;  
}
```

- in jedem Durchgang wird  $r - l$  halbiert  $\rightarrow \log_2$  Schritte
- Aufwand:  $O(\log_2)$

# Suche in mehreren Listen



## Fragen:

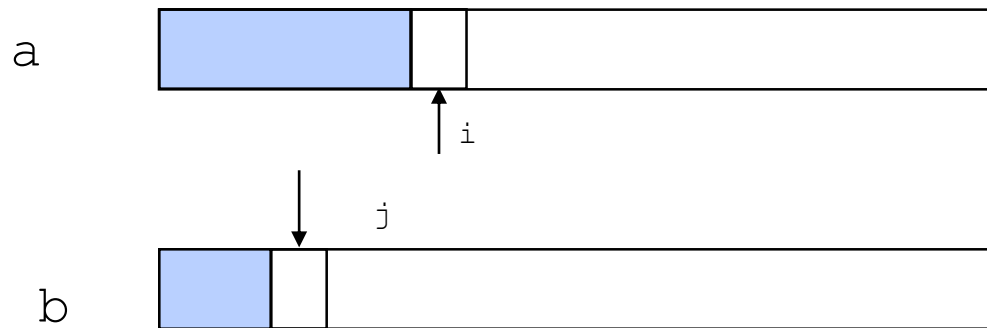
- Welche der 300 Reichsten der Schweiz beziehen Krankenkassenvergünstigung?
- Welcher Student, dessen Eltern reich sind, bekommt ein Stipendium?
- Welcher Facebook Benutzer ist auf XING und hat **Das Kapital** von Karl Marx bei Amazon gekauft?



```
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  $2 * 10000$  Elementen  $\rightarrow 10^8$

# Besserer Algorithmus wenn a und b sortiert



- **Invariante:**  $\forall k, n; k < j, n < i; b[k] \neq a[n]$
- Invariante bleibt erhalten
- Bereich für den die Invariante gilt sukzessive erweitern
- $b[j] < a[i] \rightarrow \forall k; k \leq j; b[k] < a[i] \rightarrow j \text{ um } 1 \text{ erhöhen}$
- $b[j] > a[i] \rightarrow \forall n; n \leq i; b[j] > a[n] \rightarrow i \text{ um } 1 \text{ erhöhen}$

# Schnelle Suche in zwei Arrays

```
static int indexOf(String[] a, String[] b) {  
    int i = 0, j = 0;  
    //{inv && 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++;  
    }  
    // {inv && (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

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

## ■ sortierter Array

- Einfügen  $O(n/2)$
- binäres Suchen  $O(\log_2(n))$



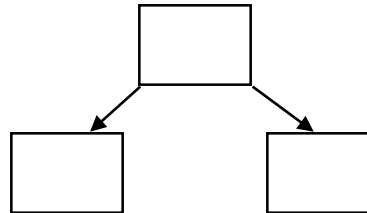
## ■ lineare sortierte Liste

- Einfügen  $O(n/2)$
- Suchen  $O(n/2)$



## ■ sortierter Binärbaum

- Einfügen  $O(\log_2(n))$
- Suchen  $O(\log_2(n))$



Frage:

- Gibt es ein Verfahren, dessen Aufwand unabhängig von der Anzahl Elemente ist.

# Schlüssel & Hashing

Gegeben sei eine Menge von Datensätzen der Form

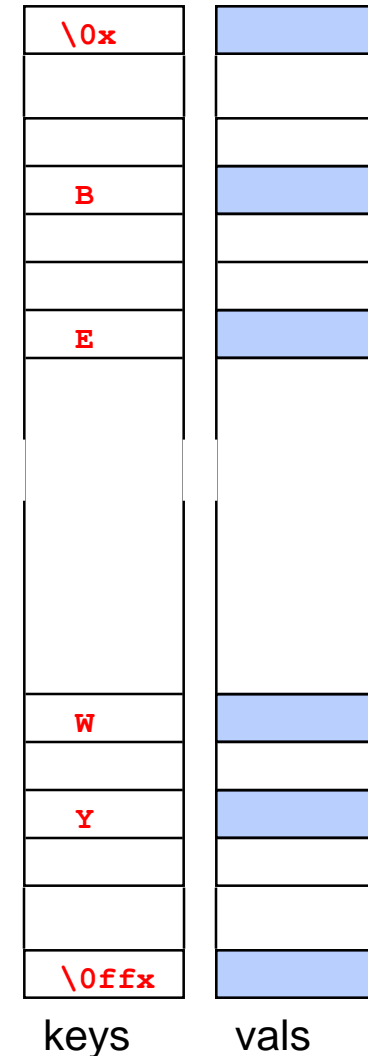


- Der Schlüssel kann ein Teil des Inhaltes sein
- Der Schlüssel besteht im einfachsten Fall aus einem String oder einem numerischen Wert.
- Mittels dem Schlüssel kann der Datensatz wiedergefunden werden.
- Bsp:
  - AHV-Nummer - Personen
  - Matrikel-Nummer- Studenten
- Aufgabe: Es sollen Daten (Objekte)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

# Menge der möglichen Schlüsselwerte klein

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

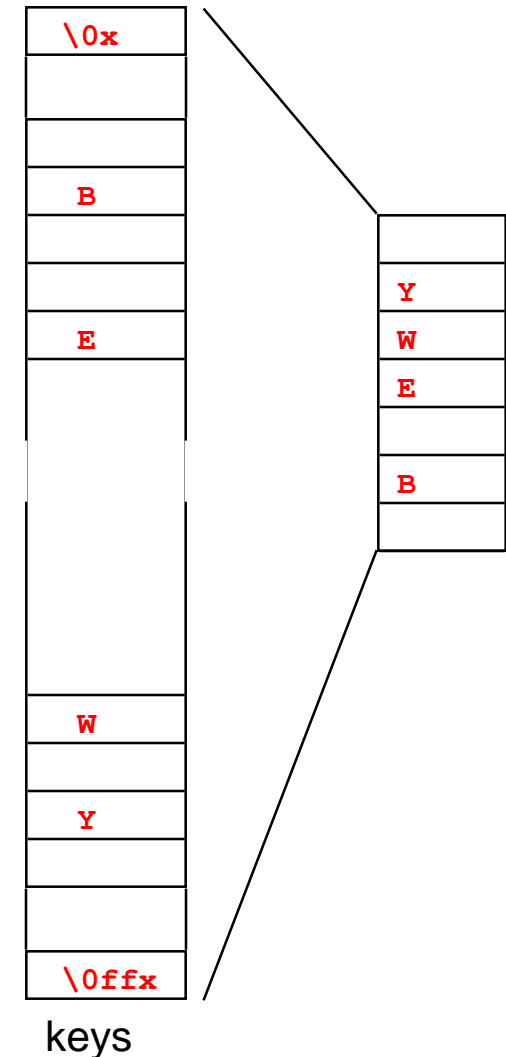
- Wertebereich A..Z
- Werte B, E, W, Y
- Einfach `char[] h = new char[256];`
- Einfügen `h[c] = c;`
- Suchen: `c = h[c];`
- Aufwand
  - Einfügen  $O(1)$
  - Suchen  $O(1)$



# Idee : Hashing

Problem: 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 abbildet.
  - Einfache Funktion:  $X \text{ modulo } \text{tableSize}$   
-> eine Zahl zwischen 0 und  $\text{tableSize}-1$
  - Eine solche Funktion nennt man **Hash Funktion**.





## Problem 1:

- Aus Hash-Wert kann der ursprüngliche Wert nicht mehr bestimmt werden, i.e.  $\notin h^{-1}(k)$

|

## Lösung

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

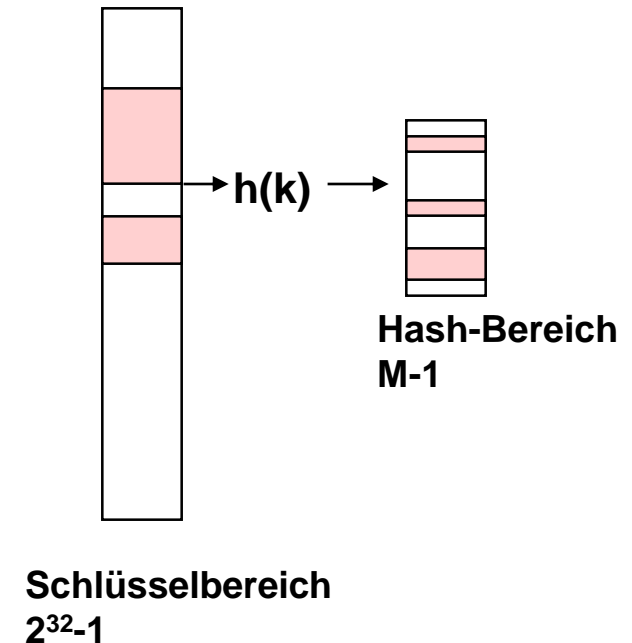
## Problem 2:

- Zwei unterschiedliche Objekte können den gleichen Hash Wert haben. d.h. sie müssten an der gleichen Stelle gespeichert werden ⇒ **Kollision**

## Lösung:

- Kollision werden vermieden - oder verringert
- Kollision wird aufgelöst
- verschiedene Verfahren zur Auflösung (später):
  - *linear/quadratic probing, ...*

- Der grosse Schlüsselbereich wird mittels der Hash-Funktion auf einen kleinen Bereich abgebildet
- Problem
  - **Massierungen (Clustering)** im (Schlüssel-) Wertebereich -> kann zu gleichen Hash-Werten führen  $\Rightarrow$  **Kollisionen**
- Hash = Durcheinander
  - die Hash-Funktion bringt den Schlüssel so durcheinander, dass er möglichst **gleichmässig** auf den ganzen Wertebereich des Schlüssels abgebildet wird
- Wirklich gute Hashfunktionen sind nicht immer einfach zu finden
- Zwei gebräuchliche Hashfunktionen
  - $h(k) = k \% M \mid M \in \text{Primzahl}$
  - $h(k) = (k * N) \% M \mid N, M \in \text{Primzahlen}$ 
    - Primzahl  $M \leq \text{Schlüsselbereich}$



# Hashtable ohne Kollisionen

```
public class Hashtable {
    final int MAX = 100;
    final int INVALID = Integer.MINVALUE; // keys array noch mit diesem Wert initialisieren
    int[] keys = new int[MAX];
    int[] vals = new int[MAX];

    private int h(int key) {
        return key * 13 % 97;
    }

    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;
    }
}
```

Hash Funktion

Überprüfe ob Feld frei

Speichere Zahl

Überprüfe ob Schlüssel korrekt

Hole Zahl

# Hashing von Strings



\*1   \*256   \*256<sup>2</sup>   \*256<sup>3</sup>

B	A	U	M
---	---	---	---

- Hashfunktion = ASCII Wert  $\times$  Position  $256^n$
- Es entstehen sehr grosse Zahlen:
  - Ein vier Zeichen langer String führt bereits zu einer Zahl in der Grössenordnung von  $256^4 = 2^{32}$ , was wenn Zeichen 32-Bit-Integer ist ( $4'294'967'296^4 = 2^{128} = 3.4 \cdot 10^{38}$ )
- In der Praxis werden deshalb Polynome zur Umwandlung von Strings verwendet (Horner Schema):  $A_3x^3 + A_2x^2 + A_1x^1 + A_0x^0$ 
  - kann als  $((A_3)x + A_2)x + A_1)x + A_0$  gerechnet werden. Trotzdem bleibt das Overflow-Problem.
- Modulo Arithmetik
  - $(a + b) \bmod x = a \bmod x + b \bmod x$
- In Java/C automatisch durch nicht behandelten Überlauf
  - bei Überlauf in Java kehrt sich das Vorzeichen bei der Interpretation des Bitmusters als *Signed* (!)

# Die hashCode Methode von Object

- 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 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. Hashwerte 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

- Bei Klassen kann der Hashcode aus den Werten der Felder berechnet werden

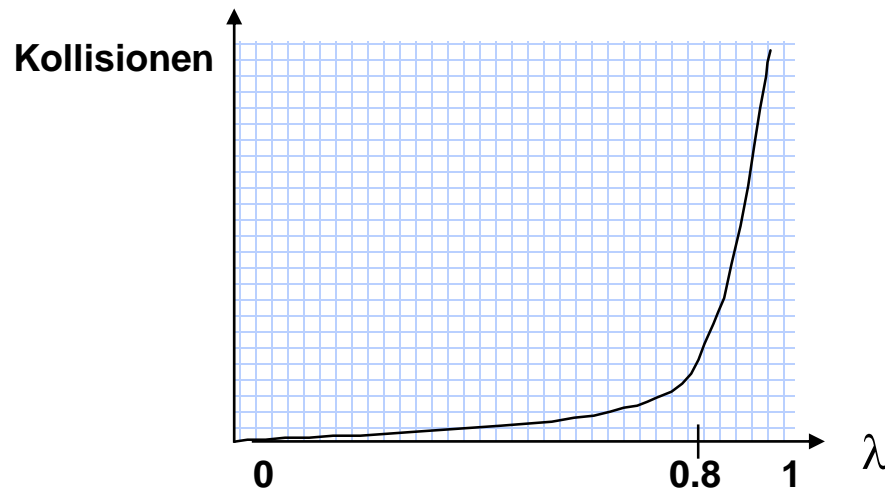
```
public class Employee {  
    int        employeeId;  
    String      name;  
    Department dept;  
  
    @Override  
    public int hashCode() {  
        int hash = 1;  
        hash = hash * 13 + employeeId;  
        hash = hash * 17 + name.hashCode();  
        hash = hash * 31 + (dept ==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

Ohne Modulo, da abhängig von Anwendung d.h. Grösse der Hashtabelle

Primzahl

# Kollisionen

- Anzahl Kollisionen hängt von der Güte der Hash-Funktion und der Belegung der Zellen ab
- Der Load-Faktor  $\lambda$ 
  - sagt wie stark der Hash-Bereich belegt ist
  - bewegt sich zwischen 0 und 1.
  - # Kollisionen ist abhängig vom Load-Faktor  $\lambda$  und der Hash-Funktion  $h$ :  $f(h, \lambda)$

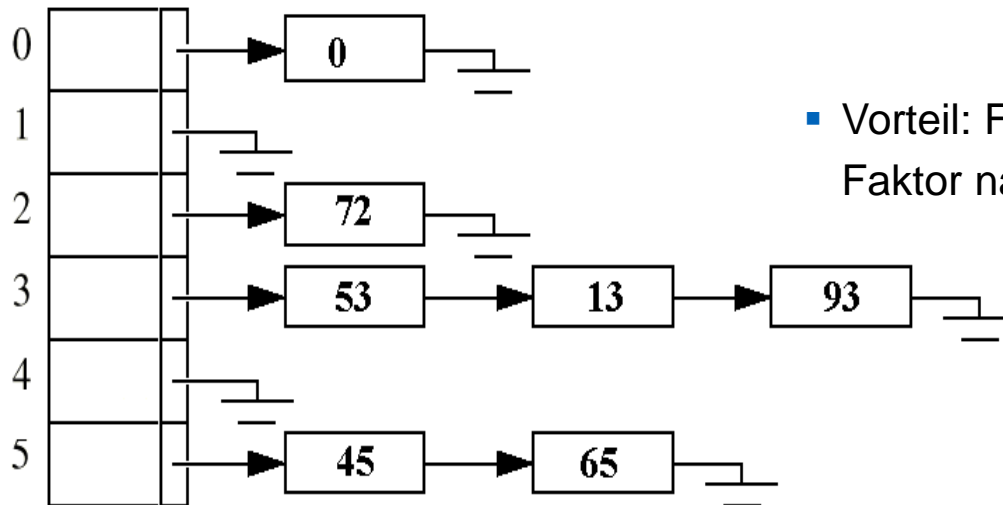




# Kollisionsauflösung 1 : Separate Chaining

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 nahe (oder grösser) als 1



Gegeben sind:

- eine Hashtabelle der Grösse 10
- eine Hash-Funktion  $h(x) = x \bmod 10$
- Input: 4371, 1323, 6173, 4199, 4344, 9679, 1989.

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

**Open Addressing:** Techniken, wo bei Kollision eine freie Zelle sonstwo in der HashTable gesucht wird.

⇒ Setzt einen Load-Faktor  $< \sim 0.8$  voraus.

- **lineares Sondieren (*Linear Probing*):**

sequentiell nach nächster freier Zelle suchen  
(mit *Wrap around*).

- **quadratisches Sondieren (*Quadratic Probing*):**

in wachsenden Schritten der Reihe nach  $F+1, F+4, F+9, \dots, F+i^2$  prüfen  
(mit *Wrap around*).

# Linear Probing 1

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

```
hash( 89, 10 ) = 9  
hash( 18, 10 ) = 8  
hash( 49, 10 ) = 9  
hash( 58, 10 ) = 8  
hash(  9, 10 ) = 9
```

- 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 HashWert gesucht bis Wert gefunden oder leere Zelle.

*After Insert 89   After Insert 18   After Insert 49   After Insert 58   After Insert 9*

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

## 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.

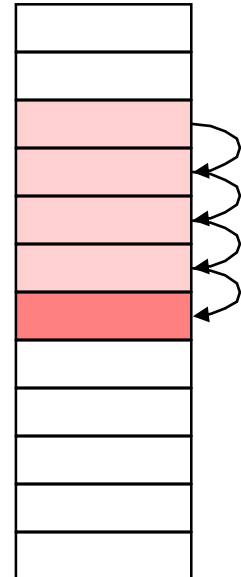
## Folge:

- Verlängerung des durchschnittlichen Zeitaufwandes zum Sondieren
- erhöhte Wahrscheinlichkeit, dass weitere Sondieren nötig wird

⇒ Bei hohem Load-Faktor oder ungünstigen Daten bricht die Performance ein!

# Linear Probing 3

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



zur Bestimmung einer  
Ausweichzelle wird einfach die  
nächste genommen ->primary  
Clustering Phänomen

Gegeben sind:

- eine Hashtabelle der Grösse 10
- eine Hash-Funktion  $H(X) = X \bmod 10$
- Input: 4371, 1323, 6173, 4199, 4344, 9679, 1989.

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

# Quadratic Probing 1

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

```
hash( 89, 10 ) = 9
hash( 18, 10 ) = 8
hash( 49, 10 ) = 9
hash( 58, 10 ) = 8
hash(  9, 10 ) = 9
```

- Hash-Funktion Input modulo Tabellengrösse

- jetzt bleiben Lücken in Hash-Tabelle offen

- die zuletzt eingefügte 9 findet ihren Platz unbeeinflusst von der zuvor eingefügten 58.

	After Insert 89	After Insert 18	After Insert 49	After Insert 58	After Insert 9
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

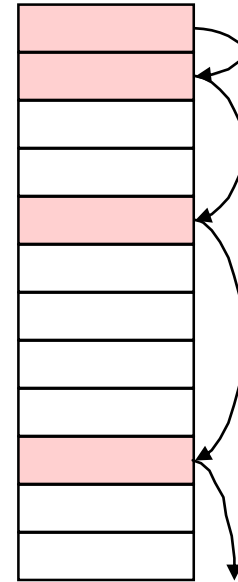


# Quadratic Probing 2

```
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 *primary*  
*Clustering* weniger auftritt.

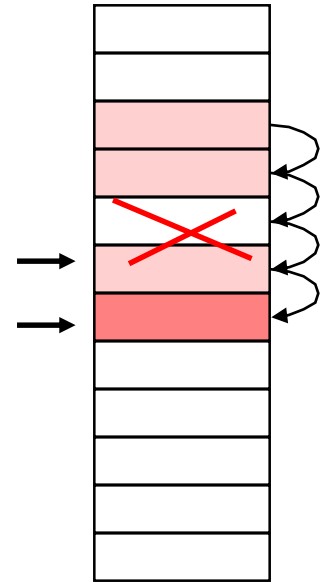
Gegeben sind:

- eine Hashtabelle der Grösse 10
- eine Hash-Funktion  $H(X) = X \bmod 10$
- Input: 4371, 1323, 6173, 4199, 4344, 9679, 1989.

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

# Löschen in Hashtabellen

- 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



## Vorteile

- Suchen Einfügen in Hash-Tabellen sehr effizient
- "Einfache" Binären Bäumen 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 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 die **ungefähre (maximale) Anzahl** bekannt ist.

# Extendible Hashing

# Platz in Hashtabelle reicht nicht mehr

- 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

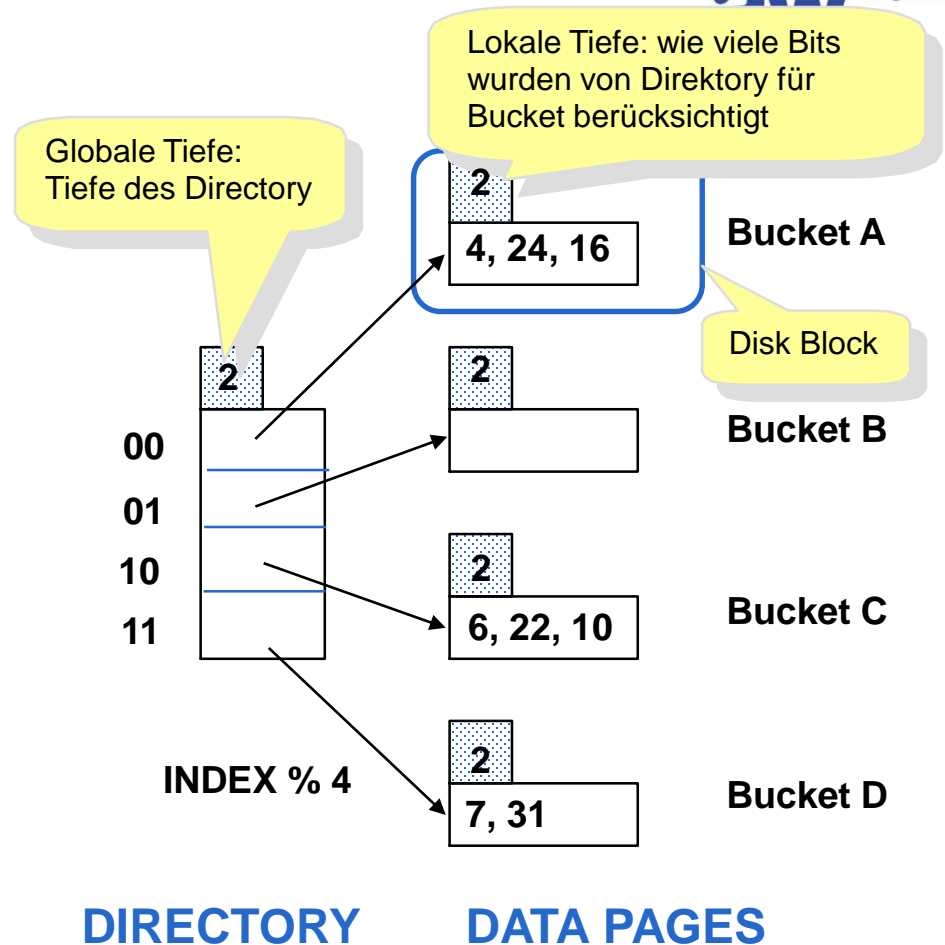
## Hashtabelle passt nicht mehr in den Hauptspeicher

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

- 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 Verzeichnis von Verweisen zu "Buckets" (=Behälter)
  - Grosse Buckets: z.B. Bucket Size = Disk Block (2048) -> kleine Schlüssel im Verzeichnis
- Verzeichnis enthält lediglich letzten n-Bits des Schlüssels und ist wesentlich kleiner (als alle Behälter zusammen)
  - kann in den Hauptspeicher geladen werden
  - kann einfacher verdoppelt werden
- Hashfunktion wird entsprechend der berücksichtigten Bits im Verzeichnis angepasst

# Beispiel

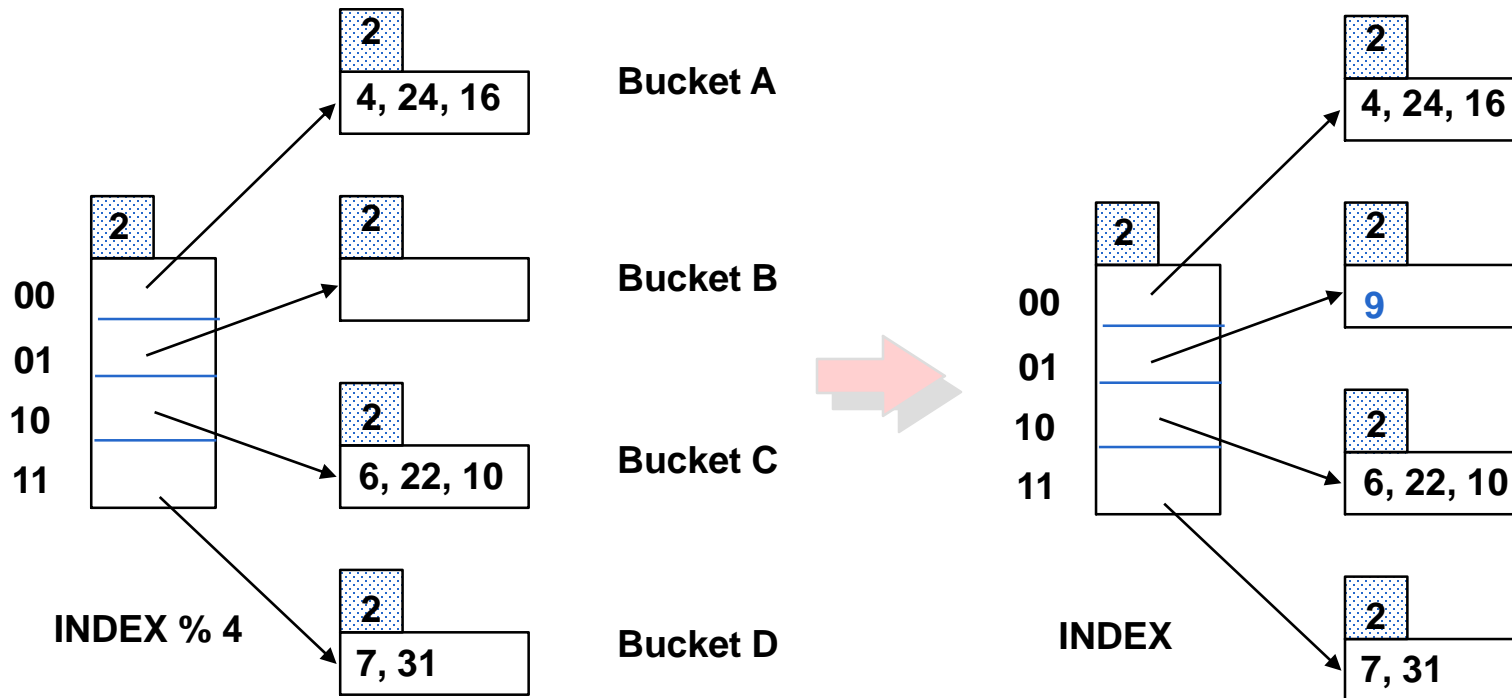
- Verzeichnis ist Array der grösse 4.
- Um das Bucket zu finden, suche im Index die letzten (*global depth* # bits of  $h(r)$ );
  - Bsp: Wert = 6 binary 110; dann ist Zeiger auf Bucket im Verzeichniseintrag 10
  - $= 6 \% 4$



- **Insert:** falls Bucket voll, dann *split* (*neue Seite und neu verteilen der Werte*).
- Falls notwendig, das Verzeichnis verdoppeln; kann bestimmt werden, durch den Vergleich von globaler zu lokaler Tiefe



# Einfügen des neuen Eintrags 9

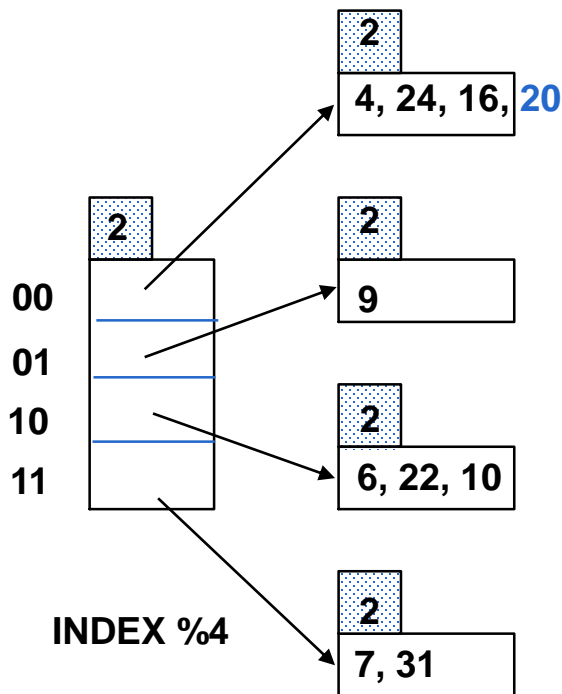


<http://www.youtube.com/watch?v=TtkN2xRAgv4>

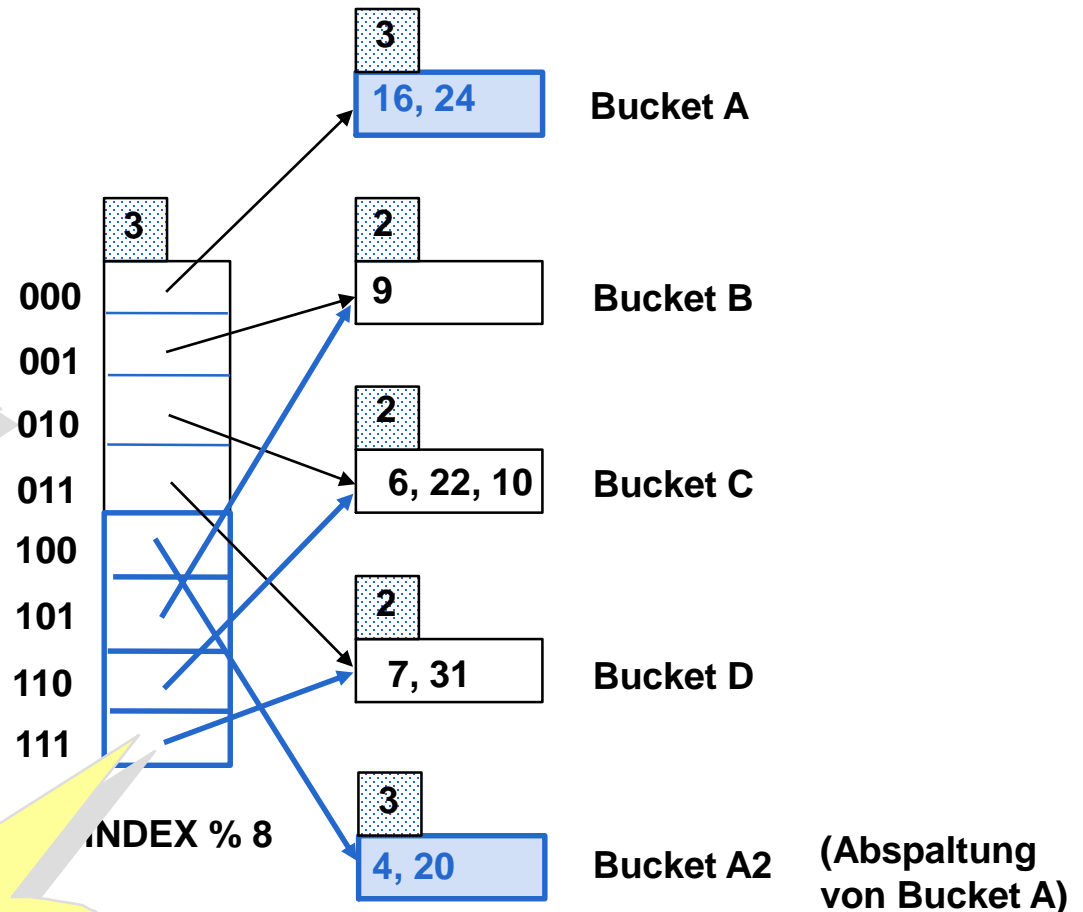
# Einfügen des neuen Eintrags 20

Bucket überlaufen mit lokaler Tiefe = globaler Tiefe

a) Directory verdoppeln!



b) Werte 4, 24, 16, 20 neu verteilen ( $\%8$ )

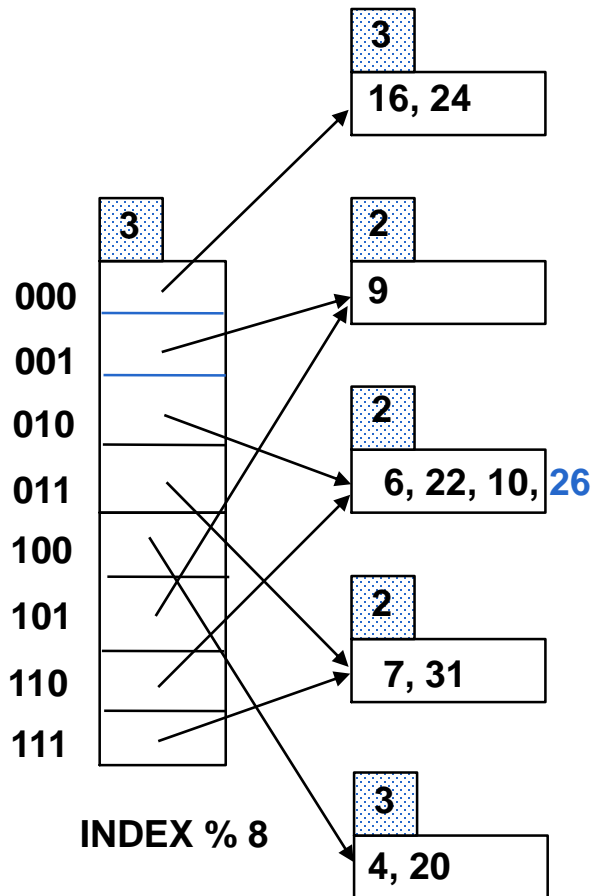


Zeiger auf nicht überlaufende Buckets verdoppelt

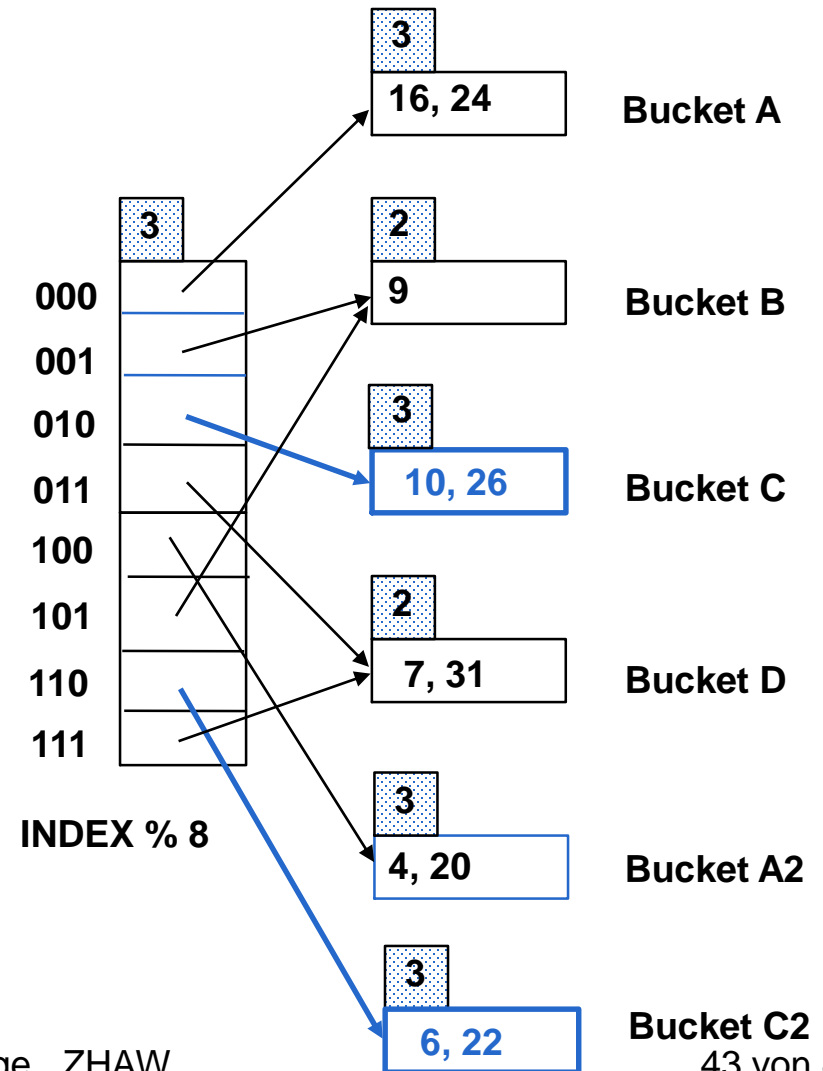
# Einfügen des neuen Eintrags 26

Bucket überlaufen mit lokaler Tiefe < globaler Tiefe

a) Bucket aufteilen



b) Werte 6, 22, 10, 26 neu verteilen (%8)

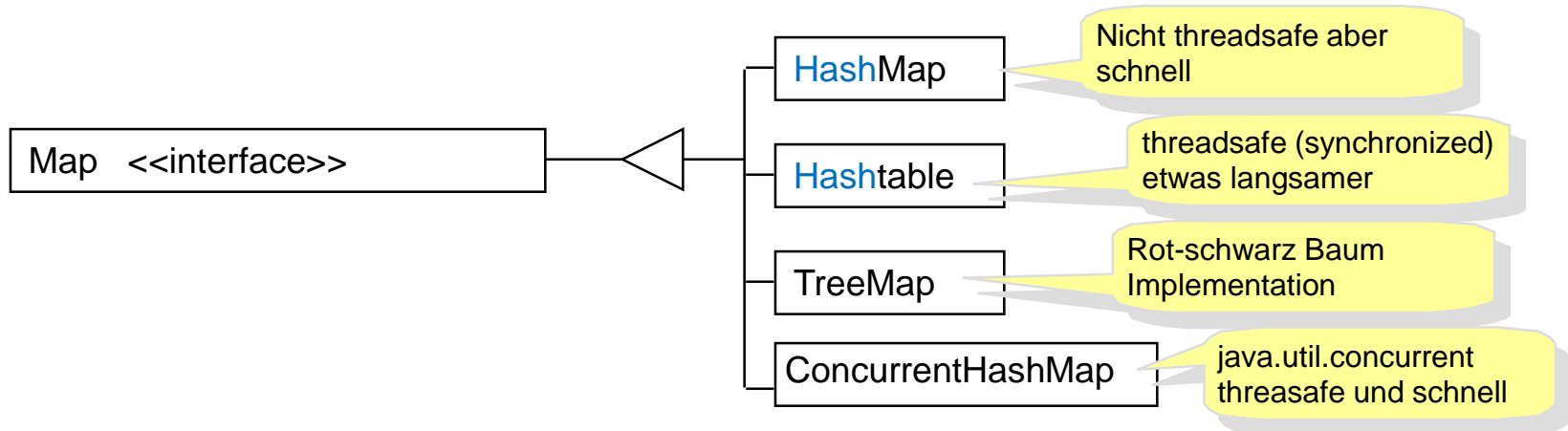


# Implementation in Java

- Hashcode Bestimmung wird dem Object überlassen

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

- Maps können auf unterschiedliche Arten implementiert



# Map<K, V> Interface

```
void clear()  
int size()  
V put(K key, V value)  
V get (Object key)  
V remove(Object key)  
boolean containsKey(Object key)  
boolean containsValue(Object value)
```

```
Collection<V> values()  
Set<K> keySet()
```

Löschen aller Elemente

Anzahl Elemente

Einfügen eines Elementes

Finden eines Elementes

Löschen eines Elementes

ist Element mit Schlüssel in Table

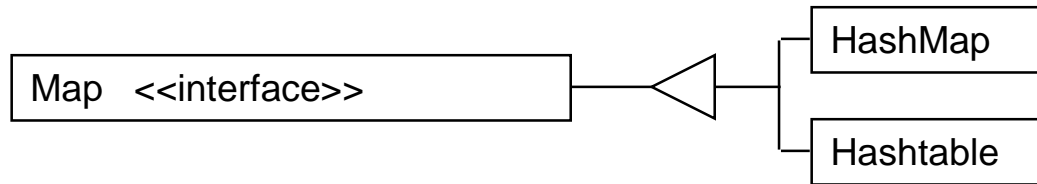
hat ein Element den Wert

alle Werte

alle Schlüssel als Set

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

# HashMap und Hashtable Implementation



- Hashtable seit Java 1.0, synchronisiert, null-Werte nicht erlaubt
- HashMap seit Java 1.2, nicht synchronisiert: null-Werte erlaubt

Für neue Programme nur noch HashMap verwenden:

If a thread-safe implementation is not needed, it is recommended to use HashMap in place of Hashtable. If a thread-safe highly-concurrent implementation is desired, then it is recommended to use ConcurrentHashMap in place of Hashtable.

(<https://docs.oracle.com/en/java/javase/11/>)

## Konstruktoren

`HashMap<K, V> ()`

Erzeugen von HashMap

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

Erzeugen von HashMap mit Grösse

## ■ Suche

- Einfache Suche
- Invariante
- Binäre Suchen
- Suche in zwei Sammlungen
- Vergleich der verschiedenen Datenstrukturen

## ■ Hashing

- Idee
- Hashfunktion
  - *gute Hashfunktionen*
- Kollisionsauflösung
  - *Überlauflisten*
  - *lineares Sondieren*
  - *quadratisches Sondieren*
- Vor- und Nachteile