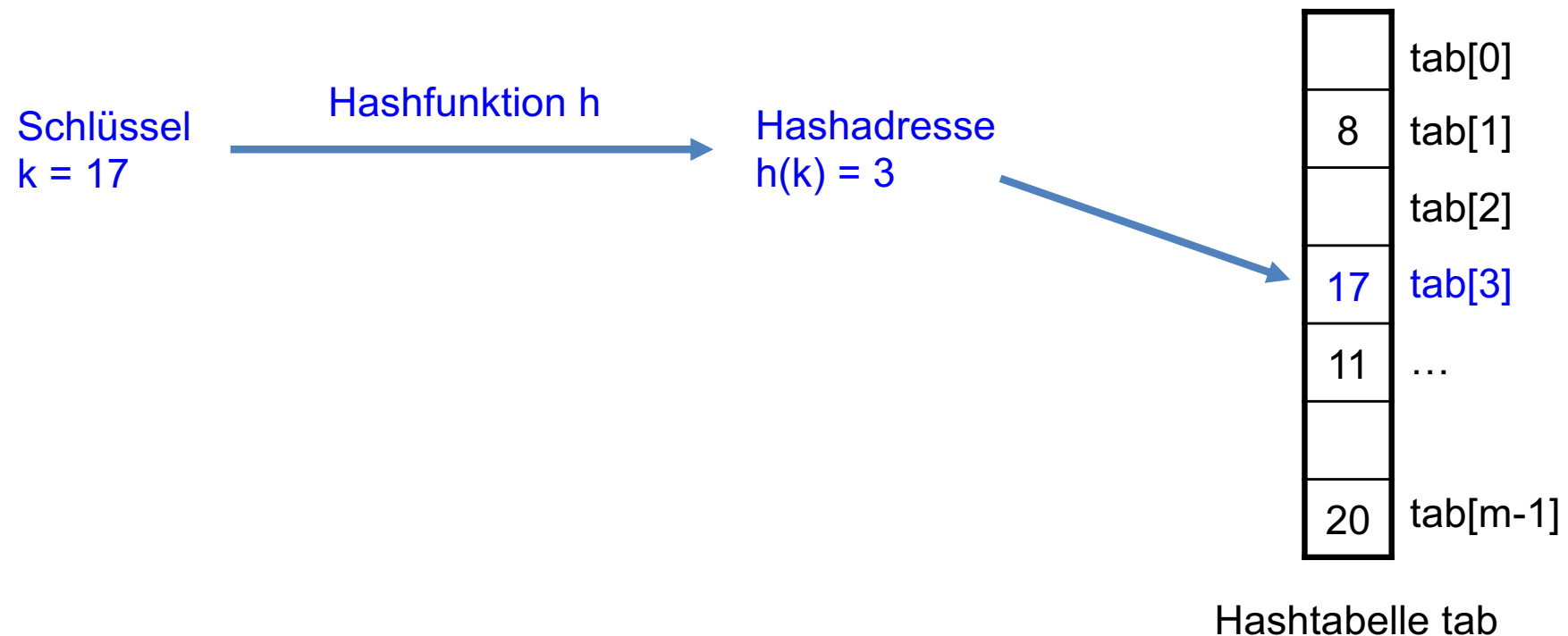


## 2. Suchen mit Hashverfahren

- Idee
- Hashfunktion
- Hashverfahren mit linear verketteten Listen
- Offene Hashverfahren
- Dynamische Hashverfahren
- Hashverfahren in Java

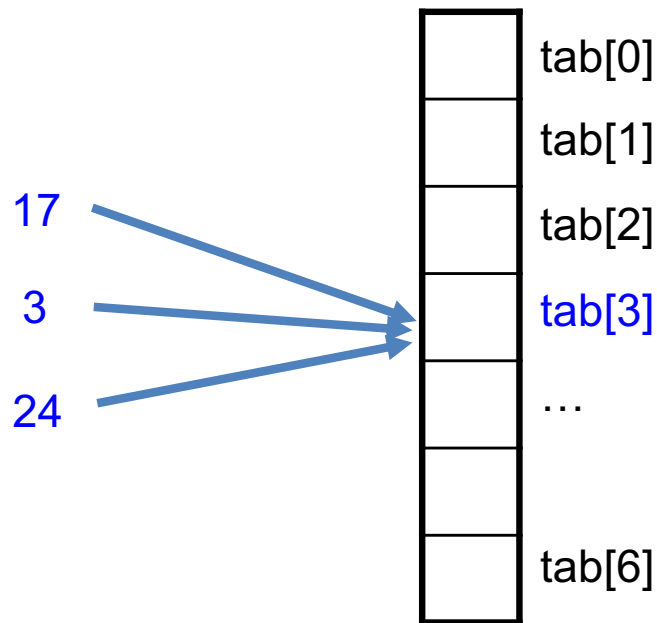
# Idee der Hashverfahren

- Mit einer Hashfunktion  $h$  wird aus einem Schlüssel  $k$  eine ganzzahlige Hashadresse  $h(k)$  mit  $0 \leq h(k) < m$  berechnet.
- Die Hashadresse gibt den Index in einem Feld der Größe  $m$  an, wo der Datensatz abgespeichert werden kann bzw. abgespeichert ist.
- Das Feld wird auch Hashtabelle genannt.



# Wichtige Anforderungen an Hashfunktionen

- Hashfunktion  $h$  sollte **einfach** zu berechnen sein.
- Gute Streuwirkung:
  - vorkommende Schlüssel sollten sich möglichst gut über die Tabelle verteilen.
  - also möglichst **wenig Adresskollisionen**



- Einfache Hashfunktion:  
 $h(k) = k \bmod m$  mit  $m = 7$
- Hashfunktion bildet jedoch Schlüssel  $k = 17, 3$  und  $24$  auf gleiche Adresse 3 ab.
- → **Adresskollision**

# Hashfunktion: Divisions-Rest-Methode

---

$$h(k) = k \bmod m;$$

- $k$  ist Schlüssel
- $m$  ist Tabellengröße
- $m$  sollte möglichst Primzahl sein

## Beispiel

7	tab[0]
8	tab[1]
	tab[2]
24	tab[3]
	tab[4]
5	tab[5]
	tab[6]

- Füge 7, 24, 5, 8 in eine Hashtabelle der Größe  $m = 7$  ein.

# Hashfunktion: Multiplikative Methode

$$h(k) = \lfloor m * \underbrace{(k\phi^{-1} - \lfloor k\phi^{-1} \rfloor)}_{\in [0, 1)} \rfloor$$

$\underbrace{\hspace{10em}}_{\in \{0,1, \dots m-1\}}$

- k ist Schlüssel
- m ist Tabellengröße
- $\phi^{-1} = (\sqrt{5} - 1)/2 \approx 0.6180339887$  ist Kehrwert des goldenen Schnitts.
- $\lfloor x \rfloor$  rundet auf die nächste kleinere ganze Zahl ab.

## Beispiel

tab[0]	5
[1]	10
[2]	2
[3]	7
[4]	4
[5]	9
[6]	1
[7]	6
[8]	3
[9]	8

- Füge 1, 2, 3, ..., 10 in eine Hashtabelle der Größe m = 10 ein.
- Alle Elemente werden **ohne Kollision** verteilt!

# Bei Division-Rest-Methode sollte Tabellengröße Primzahl sein

---

## Beispiel Personaldaten –

Wahl einer geradzahligen Tabellengröße erweist sich als ungeschickt!

- Die Personaldaten einer Firma sollen in einer Hashtabelle abgespeichert werden.
- Ganzzahlige Personalnummer als Schlüssel:
  - männliche Person:  $xx \dots x0$
  - weibliche Person:  $xx \dots x1$
- Hashfunktion  $h = k \bmod m$  mit einer geraden Tabellengröße  $m$
- Ungleichmäßige Streuung bei ungleicher Geschlechterverteilung:
  - männliche Person:  $h(x \dots xx0)$  ist gerade
  - weibliche Person:  $h(x \dots xx1)$  ist ungerade

# Bei Division-Rest-Methode sollte Tabellengröße Primzahl sein

---

## Beispiel Personaldaten –

Wahl einer Tabellengröße von  $m = 1000$  erweist sich als ungeschickt!

- Die Personaldaten einer Firma sollen in einer Hashtabelle abgespeichert werden.
- **Personalnummer** ist 6-stellig, wobei die hintersten 3 Stellen die Abteilung codieren, in der die Person arbeitet.

xxx**yyy**  
└─┬─┘  
Abteilung

- Hashfunktion  $h = k \bmod m$  mit **Tabellengröße  $m = 1000$**
- **Ungleichmäßige Streuung** bei ungleicher Verteilung der Mitarbeiter auf die einzelnen Abteilungen.

$$h(\text{xxx}\text{yyy}) = \text{yyy}$$

# Einschub: Verteilung der Primzahlen

## Primzahlfunktion

$\pi(n)$  = Anzahl der Primzahlen  $\leq n$ .

## Primzahlsatz

$$\pi(n) \sim n/\ln(n)$$

Damit gilt für die Primzahlhäufigkeit:

$$\pi(n)/n \sim 1/\ln(n)$$

## Tabelle zeigt die tatsächliche und die geschätzte Primzahlhäufigkeit

n	$\pi(n)/n$	$1/\ln(n)$
$10^4$	0.123	0.11
$10^5$	0.096	0.087
$10^6$	0.078	0.072
$10^7$	0.066	0.062
$10^8$	0.058	0.054
$10^9$	0.051	0.048
$10^{100}$	?	0.0043
$10^{200}$	?	0.0021

Im praktisch relevanten Bereich liegt die Primzahlhäufigkeit bei 5 bis 10%.



# Hashfunktion für Strings als Schlüssel

- Interpretierte String  $s = z_0 \dots z_{n-2} z_{n-1}$  (Folge von ASCII-Zeichen) als eine Zahl im Stellenwertsystem 31:

$$h(z_0 \dots z_{n-2} z_{n-1}) = [z_0 * 31^{n-1} + \dots + z_{n-2} * 31^1 + z_{n-1} * 31^0] \bmod m;$$

- 31 ist eine Primzahl und führt zu einer guten Streuung.
- Mithilfe des Horner-Schemas lässt sich der Ausdruck effizienter berechnen:

$$h(z_0 \dots z_{n-2} z_{n-1}) = [( \dots (z_0 * 31 + z_1) * 31 + \dots + z_{n-2}) * 31 + z_{n-1}] \bmod m;$$

- Bei einer Implementierung der Hashfunktion kann ein evtl. auftretender Überlauf ignoriert werden. Die Hashadresse kann dabei negativ werden, was abgeprüft werden muss.

```
static int hash(String key){
    int adr = 0;
    for (int i = 0; i < key.length(); i++)
        adr = 31*adr + key.charAt(i);

    if (adr < 0)
        adr = -adr;
    return adr % m;
}
```

# Hashfunktion hashCode in Java (1)

- In Java ist in der Klasse `Object` die Methode `hashCode` definiert, die jedes Objekt auf einen ganzzahligen Wert abbildet.

```
class Object {  
    public int hashCode() {...}  
    // ...  
}
```

- Für alle Wrapper-Klassen wie `Integer`, `Long`, `Short`, etc. und für die Klasse `String` ist `hashCode` geeignet überschrieben.
- `hashCode` für `String` ist ähnlich implementiert wie die for-Schleife auf der vorhergehenden Seite.

Es ist zu berücksichtigen, dass `hashCode` eine negative int-Zahl zurückliefern kann. Wird der HashCode als Adresse für eine Hash-Tabelle benötigt, muss ein negativer Wert abgefangen und modulo der Tabellengröße gerechnet werden:

```
String key = "Zimmer";  
  
int adr = key.hashCode();  
if (adr < 0)  
    adr = -adr;  
adr = adr % m;
```

liefert -1618018788

# Hashfunktion hashCode in Java (2)

---

## Hashfunktion für zusammengesetzte Schlüssel

- wird für einen zusammengesetzten Schlüsseltyp eine Hashfunktion benötigt, dann überschreibt man am besten für den Schlüsseltyp die hashCode-Methode.
- Dabei kann eine ähnliche Technik wie bei einem String-Schlüssel eingesetzt werden.
- Beispiel:

```
public class Datum {  
    private int tag;  
    private int mon;  
    private int jahr;  
    // ...  
  
    @Override public int hashCode() {  
        int adr = 0;  
        adr = 31*adr + tag;  
        adr = 31*adr + mon;  
        adr = 31*adr + jahr;  
        return adr;  
    }  
}
```

# Hashfunktion hashCode in Java (3)

---

## Hashfunktion für zusammengesetzte Schlüssel (Fortsetzung)

- Im allgemeinen ist auch hier zu berücksichtigen, dass ein Überlauf auftreten kann und hashCode eine negative int-Zahl zurückliefert.
- Wird der hashCode als Adresse für eine Hash-Tabelle benötigt, muss ein negativer Wert abgefangen und modulo der Tabellengröße gerechnet werden.
- Beispiel:

```
Datum key = new Datum(24,12,2011);  
int adr = key.hashCode();  
if (adr < 0)  
    adr = -adr;  
adr = adr % m;
```

ist in diesem Fall unnötig.

Bei Datum.hashCode kann es keinen Überlauf geben.

# Adresskollision

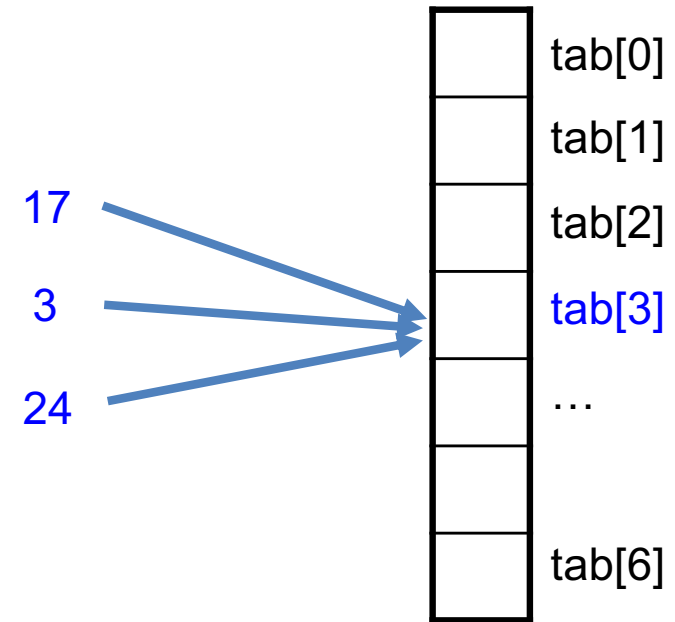
- Im allgemeinen ist eine Hashfunktion nicht injektiv. D.h. unterschiedliche Schlüssel können die gleiche Hashadresse haben.

- Beispiel: mit  $h(k) = k \bmod m$  und  $m = 7$  gilt:

$$h(17) = h(3) = h(24) = 3.$$

- Die Wahrscheinlichkeit einer Adresskollision ist sehr groß. Sogar bei einer sehr gut streuenden Hashfunktion und einer im Vergleich zur Schlüsselzahl großen Hashtabelle. (siehe Geburtstagsparadoxon auf nächste Folie)

- Ansätze zur Kollisionsbehandlung:
  - kollidierende Datensätze werden in **linear verketteten Listen** gehalten.
  - **Offene Hashverfahren:**  
bei Kollision in der Tabelle nach freien Stellen sondieren



# Einschub: Geburtstagsparadoxon

---

- In einer Gruppe von 23 Personen ist es wahrscheinlich (nämlich Wahrscheinlichkeit  $p \approx 0.51$ ), dass 2 Personen am gleichen Tag Geburtstag haben.

- Begründung:

Wahrscheinlichkeit  $\bar{p}$ , dass alle  $n = 23$  Personen an unterschiedlichen Tagen Geburtstag haben:

$$\bar{p} = \underbrace{\frac{364}{365}}_{2.\text{Person}} * \underbrace{\frac{363}{365}}_{3.\text{Person}} * \dots * \underbrace{\frac{343}{365}}_{23.\text{Person}} \approx 0.49$$

Damit ist die Wahrscheinlichkeit  $p$ , dass wenigstens zwei Personen an einem gleichen Tag Geburtstag haben:

$$p = 1 - \bar{p} \approx 0.51$$

- Konsequenz:  
würde man die Daten von 23 Personen in eine Tabelle der Größe 365 mit einem Hashverfahren abbilden und als Hashfunktion  
 $h(x) = \text{Geburtstag der Person } x \text{ auf } \{0, 1, 2, \dots, 364\} \text{ umgerechnet}$   
wählen, dann würde es wahrscheinlich eine Kollision geben.

## 2. Suchen mit Hashverfahren

- Idee
- Hashfunktion
- Hashverfahren mit linear verketteten Listen
- Offene Hashverfahren
- Dynamische Hashverfahren
- Hashverfahren in Java

# Hashverfahren mit linear verketteten Listen (1)

## Idee

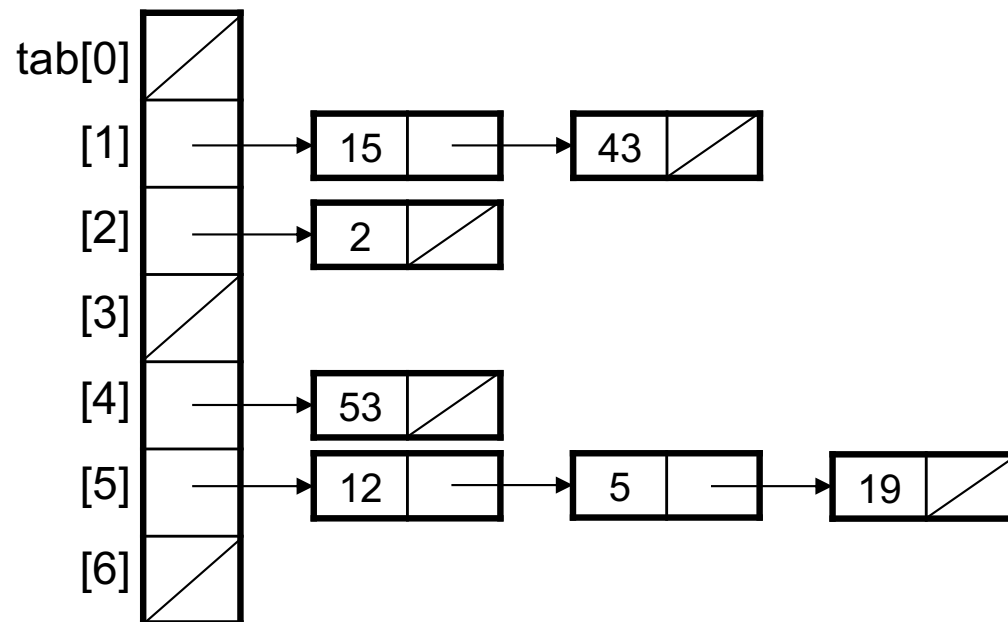
Der Hashtabellen-Eintrag  $\text{tab}[i]$  zeigt auf eine verkettete Liste, die alle Datensätze mit der gleichen Hashadresse  $i$  enthalten.

## Beispiel

Für  $m = 7$  mit  $h(k) = k \bmod m$  ergibt sich nach dem Einfügen von

12, 53, 5, 15, 2, 19, 43

folgende Hashtabelle:





# Hashverfahren mit linear verketteten Listen (2)

## Algorithmen:

```
V search (K key) {  
    suche in tab[ h(key) ] nach Schlüssel key;  
    if (key gefunden)  
        return Daten des gefunden Datensatzes;  
    else  
        return null;  
}
```

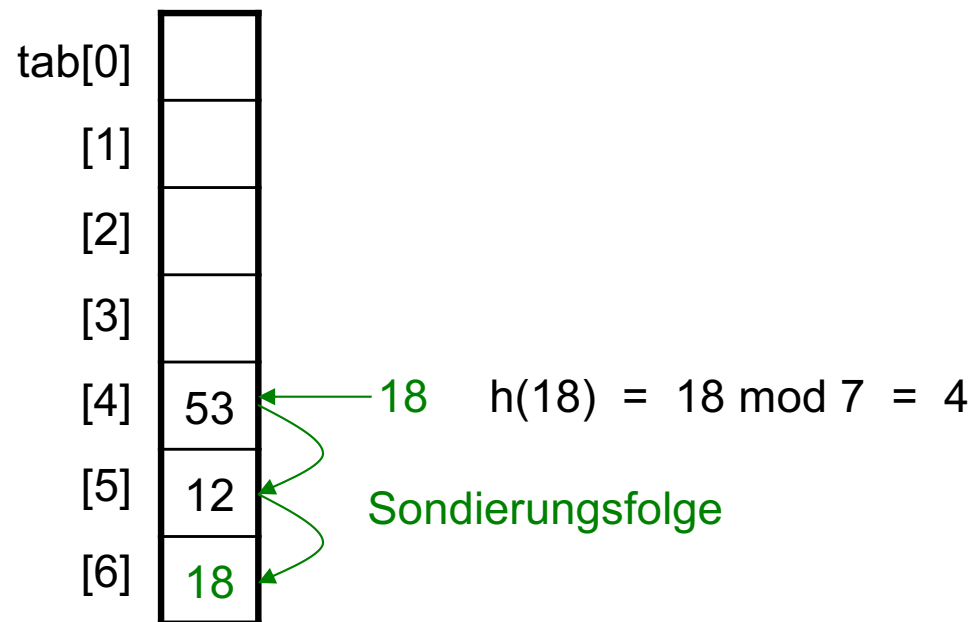
```
V insert (K key, V value) {  
    suche in tab[ h(key) ] nach Schlüssel key;  
    if (key gefunden)  
        // Schlüssel bereits vorhanden;  
        alte Daten durch value ersetzen;  
        return alte Daten;  
    else {  
        füge key am Ende oder  
        am Anfang der Liste ein;  
        return null;  
    }  
}
```

```
V remove (K key) {  
    suche in tab[ h(key) ] nach Schlüssel key;  
    if (key gefunden) {  
        entferne Knoten k aus Liste;  
        return Daten von Knoten k;  
    }  
    else  
        return null;  
}
```

# Offene Hashverfahren (1)

## Idee:

- Alle Datensätze werden in einem Feld (d.h. keine verketteten Listen) untergebracht.
- Falls beim Eintragen des Schlüssels  $k$  die Adresse  $h(k)$  bereits belegt ist, wird gemäß einer **Sondierungsfolge** die erste freie Adresse gesucht und  $k$  dort abgespeichert.



- Beim Suchen von  $k$  wird ebenfalls die Sondierungsfolge durchlaufen.
- Zu unterschiedlichen Zeitpunkten können gleiche Schlüssel auf unterschiedliche Adressen abgebildet werden; daher auch die Bezeichnung **Hashverfahren mit offener Adressierung**.

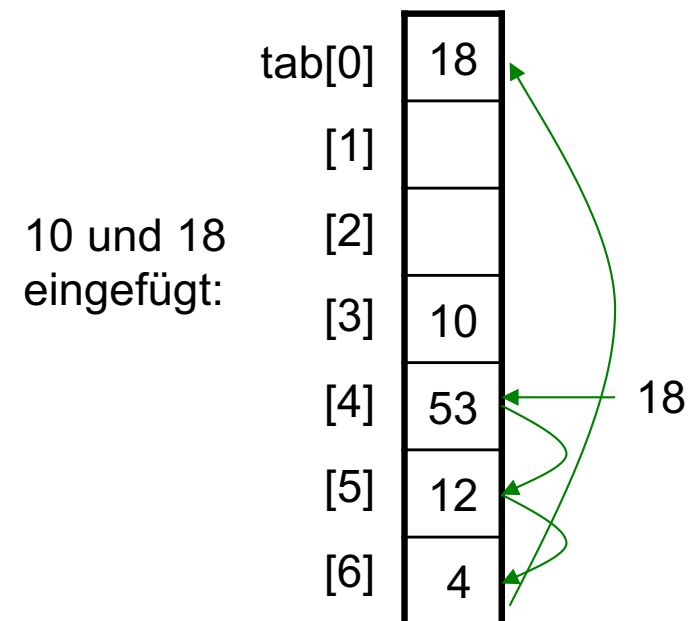
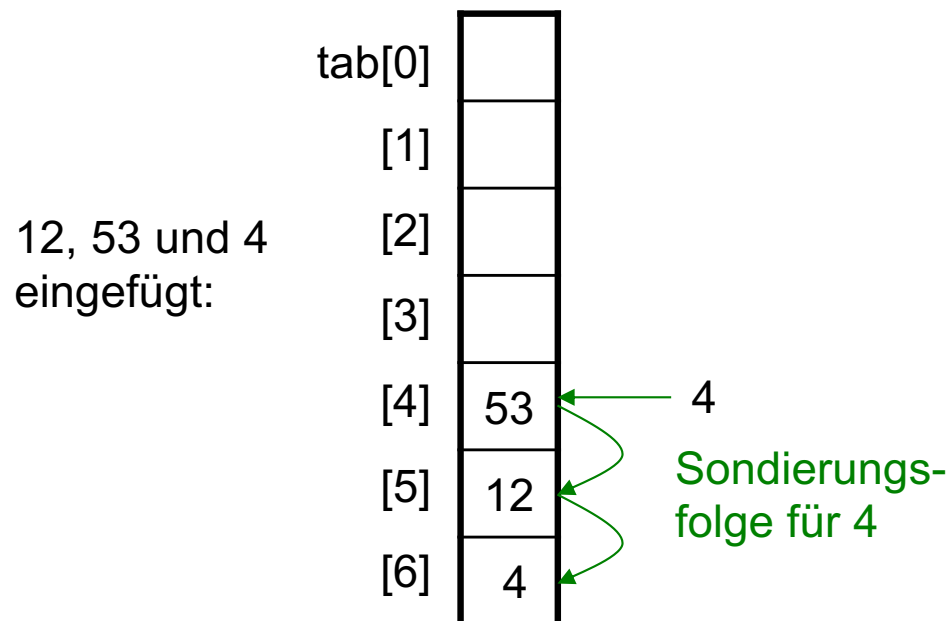
# Offene Hashverfahren (2)

## Allgemeine Sondierungsfolge

$$h(k) + s(j,k) \bmod m \quad \text{mit } j = 0, 1, \dots, m-1$$

## Beispiel (lineare Sondierungsfolge)

$$h(k) + j \bmod m \quad \text{mit } j = 0, 1, \dots, m-1 \text{ und } h(k) = k \bmod m$$



# Offene Hashverfahren (3)

## Algorithmus zum Suchen:

```
V search (K key) {  
    if ( (adr = searchAdr(key)) != -1)  
        return tab[adr].value;  
    else  
        return null;  
}
```

```
int searchAdr (K key) {  
    j = 0;  
    do {  
        adr = (h(key) + s(j,key)) % m;  
        j++;  
    } while (tab[adr] != "leer" && tab[adr].key != key);  
  
    if (tab[adr] != "leer")  
        return adr;  
    }  
    else  
        return -1;  
}
```

## Wichtig:

Um Endlosschleifen bei stark gefüllten Tabellen zu vermeiden, sind die Längen der Sondierungsfolgen zu beschränken.

# Offene Hashverfahren (4)

## Algorithmus zum Löschen:

```
V remove (K key) {  
    if ( (adr = searchAdr(key)) != -1)  
        oldValue = tab[adr].value;  
        tab[adr] = "gelöscht";  
        return oldValue;  
    }  
    else  
        return null;  
}
```

### Wichtig:

Eine Hashadresse hat 3 Zustände:

- Eintrag vorhanden
- Eintrag gelöscht
- Eintrag leer

Zu Beginn sind alle Einträge leer.

### Grund:

Beim Löschen können Lücken entstehen, die bei einer späteren Suchoperation übersprungen werden müssen.

# Offene Hashverfahren (5)

## Algorithmus zum Einfügen:

```
V insert (K key, V value) {  
    if ( (adr = searchAdr(key)) != -1) {  
        oldValue = tab[adr].value;  
        tab[adr].value = value;  
        return oldValue;  
    }  
  
    // Neueintrag:  
    j = 0;  
    do {  
        adr = (h(k) + s(j,k)) % m;  
        j++;  
    } while (tab[adr] != "leer" && tab[adr] != "gelöscht");  
    tab[adr].key = key;  
    tab[adr].value = value;  
    return null;  
}
```

Es werden zuerst wieder  
die Lücken gefüllt

# Lineares Sondieren

- $s(j,k) = j$
- Damit ergibt sich die Sondierungsfolge:

$h(k)$

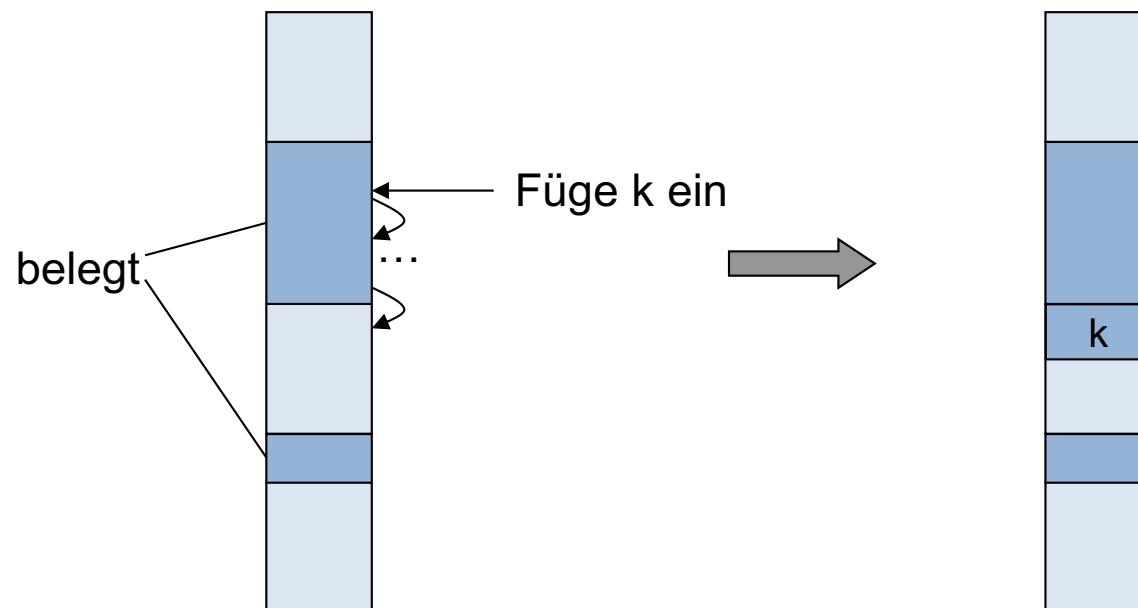
$h(k) + 1 \bmod m$

$h(k) + 2 \bmod m$

...

$h(k) + m-1 \bmod m$

- Lineares Sondieren tendiert zur **Clusterbildung**:  
große belegte Cluster haben eine stärkere Tendenz zu wachsen als kleinere.



# Einfaches quadratisches Sondieren

---

- $s(j,k) = j^2$
- Damit ergibt sich die Sondierungsfolge:

$h(k)$

$h(k) + 1 \bmod m$

$h(k) + 2^2 \bmod m$

$h(k) + 3^2 \bmod m$

...

$h(k) + (m-1)^2 \bmod m$

- Quadratisches Sondieren streut wesentlich besser als lineares Sondieren.
- Jedoch Probleme mit Erreichbarkeit aller Einträge (siehe nächste Seite).



# Problem beim quadratischen Sondieren

---

- Es wird mit einer quadratischen Sondierungsfolge im allgemeinen **nicht jede Adresse** erreicht.
- Etwas besser: wenn **m eine Primzahl** ist, dann werden mit jeder Sondierungsfolge **genau  $\lceil m/2 \rceil$  aller Einträge** erreicht. (Beweis siehe [Weiss 2007])

- **Beispiel:**

mit  $m = 16$  und  $h(k) = 0$  erreicht die Sondierungsfolge nur 4 der 16 Einträge:

$$0 \bmod 16 = 0$$

$$1^2 \bmod 16 = 1$$

$$2^2 \bmod 16 = 4$$

$$3^2 \bmod 16 = 9$$

$$4^2 \bmod 16 = 0$$

$$5^2 \bmod 16 = 1$$

$$6^2 \bmod 16 = 4$$

...

$$15^2 \bmod 16 = 1$$

- **Beispiel:**

mit Primzahl  $m = 7$  und  $h(k) = 0$  erreicht die Sondierungsfolge genau  $\lceil m/2 \rceil = 4$  der 7 Einträge:

$$0 \bmod 7 = 0$$

$$1^2 \bmod 7 = 1$$

$$2^2 \bmod 7 = 4$$

$$3^2 \bmod 7 = 2$$

$$4^2 \bmod 7 = 2$$

$$5^2 \bmod 7 = 4$$

$$6^2 \bmod 7 = 1$$

# Alternierend quadratisches Sondieren

---

- Als Tabellengröße  $m$  wird eine Primzahl der Form  $4i + 3$  gewählt.  
Z.B. 3, 7, 11, 19, 23, 31, ...

- Die alternierend quadratische Sondierungsfolge wird definiert durch:

$$s(j,k) = \lceil j/2 \rceil^2 (-1)^j$$

- Die Sondierungsfolge lautet konkret:

$$h(k)$$

$$h(k) - 1 \bmod m$$

$$h(k) + 1 \bmod m$$

$$h(k) - 2^2 \bmod m$$

$$h(k) + 2^2 \bmod m$$

...

$$h(k) - \lfloor m/2 \rfloor^2 \bmod m$$

$$h(k) + \lfloor m/2 \rfloor^2 \bmod m$$

- Aufpassen:  $\bmod$  ist mathematisch definiert und unterscheidet sich für negative Zahlen vom Modulo-Operator  $\%$  in Java.

# Einschub: symmetrische mod-Funktion

---

## Symmetrische mod-Funktion:

$$x \bmod m = x - (x/m) * m \quad x/m \text{ ist hier eine Ganzzahldivision}$$

## Beispiele:

$$11 \bmod 7 = 11 - (11/7) * 7 = 11 - 1 * 7 = 4$$

$$-11 \bmod 7 = -11 - (-11/7) * 7 = -11 - (-1) * 7 = -4$$

## %-Operator in Java und C/C++:

```
System.out.println(11 % 7);    // 4
```

```
System.out.println(-11 % 7);   // -4
```

# Einschub: mathematische mod-Funktion

---

## Mathematische mod-Funktion:

$$x \bmod m = x - \lfloor x/m \rfloor * m \quad x/m \text{ ist hier eine Gleitkommadivision}$$

## Beispiele:

$$11 \bmod 7 = 11 - \lfloor 11/7 \rfloor * 7 = 11 - 1 * 7 = 4$$

$$-11 \bmod 7 = -11 - \lfloor -11/7 \rfloor * 7 = -11 - (-2) * 7 = 3$$

## %-Operator in Python:

```
print(11 % 7);    // 4  
println(-11 % 7); // 3
```

## Funktion floorMod in Java:

```
System.out.println(Math.floorMod(11, 7));    // 4  
System.out.println(Math.floorMod(-11, 7));   // 3
```

# Alternierend quadratisches Sondieren

---

- Mit alternierend quadratischem Sondieren werden **alle Einträge** erreicht.  
(Beweis in C. E. Radtke, The use of quadratic residue search, 1970)
- **Beispiel:**  
mit Primzahl  $m = 7$  und  $h(k) = 0$  erreicht die  
Sondierungsfolge **alle 7 Einträge**:

$$0 \bmod 7 = 0$$

$$-1^2 \bmod 7 = 6$$

$$+1^2 \bmod 7 = 1$$

$$-2^2 \bmod 7 = 3$$

$$+2^2 \bmod 7 = 4$$

$$-3^2 \bmod 7 = 5$$

$$+3^2 \bmod 7 = 2$$

# Beispiel zu alternierend quadratischem Sondieren

---

Einfügen: 3, 4

	[0]
	[1]
	[2]
3	[3]
4	[4]
	[5]
	[6]

Einfügen: 10

	[0]
	[1]
10	[2]
3	[3]
4	[4]
	[5]
	[6]

Einfügen: 17

	[0]
	[1]
10	[2]
3	[3]
4	[4]
	[5]
17	[6]

Löschen: 10

	[0]
	[1]
<del>10</del>	[2]
3	[3]
4	[4]
	[5]
17	[6]

Einfügen: 24

	[0]
	[1]
24	[2]
3	[3]
4	[4]
	[5]
17	[6]

# Double Hashing

---

- Für die Sondierungsfolge wird eine weitere Hashfunktion  $h'$  gewählt.
- Die Sondierungsfolge wird definiert durch:

$$s(j,k) = j * h'(k)$$

- Damit ergibt sich folgende Sondierungsfolge:

$$h(k)$$

$$h(k) + 1 * h'(k) \bmod m$$

$$h(k) + 2 * h'(k) \bmod m$$

$$h(k) + 3 * h'(k) \bmod m$$

...

- Die beiden Hashfunktionen sollten möglichst unabhängig sein.  
Eine gute Wahl ist:

$$h(k) = k \bmod m$$

$$h'(k) = (k+1) \bmod (m-2).$$

- Außerdem ist es wichtig, dass die Tabellengröße  $m$  eine Primzahl ist.  
Nur dann erreicht jede Sondierungsfolge alle Einträge.

# Analyse der Hashverfahren

---

## Analyse im schlechtesten Fall

- Alle  $n$  Einträge haben die gleiche Hashadresse.  
Daher muss im schlechtesten Fall eine Operation eine Liste mit  $n$  Einträgen ablaufen.
- Dies ist so unwahrscheinlich, dass es praktisch nicht vorkommt.

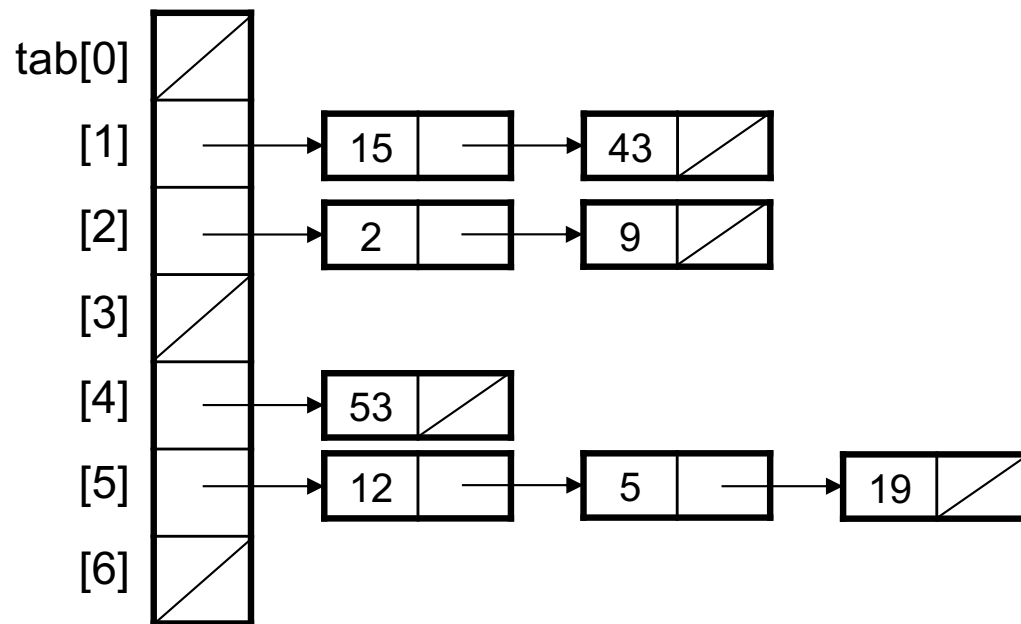
## Analyse im mittleren Fall

- Wichtige Maßzahlen:
  - $C_{\text{Erfolg}}$  Anzahl der durchschnittlich betrachteten Einträge bei **erfolgreicher Suche**.
  - $C_{\text{Nicht-Erfolg}}$  Anzahl der durchschnittlich betrachteten Einträge bei **nicht erfolgreicher Suche**.
- $C_{\text{Erfolg}}$  und  $C_{\text{Nicht-Erfolg}}$  hängen nur ab vom **Füllungsgrad (load factor)  $\alpha = n/m$** ,  
wobei  $n$  = Anzahl der Einträge und  $m$  = Tabellengröße.



# Beispiele für Füllungsgrad $\alpha$

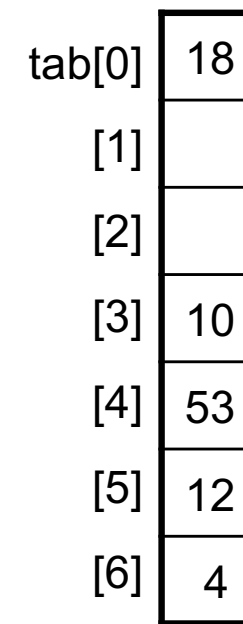
## Hashverfahren mit Verkettung



$$\alpha = n/m$$
$$= 8/7 = 1.14 = 114\%$$

$n$  = Anzahl der Einträge und  $m$  = Tabellengröße.

## Offenes Hashverfahren mit linearer Sondierung



$$\alpha = n/m$$
$$= 5/7 = 0.71 = 71\%$$

# Komplexität der Hashverfahren

Verfahren	$C_{\text{Erfolg}}$	$C_{\text{Nicht-Erfolg}}$
Hashverfahren mit Verkettung	$1 + \frac{\alpha}{2}$	$\alpha$
Offenes Hashverfahren mit linearem Sondieren	$\frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right)$	$\frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)^2} \right)$
Offenes Hashverfahren mit alternierend quadratischem Sondieren	$1 + \ln \left( \frac{1}{1 - \alpha} \right) - \frac{\alpha}{2}$	$\frac{1}{1 - \alpha} - \alpha + \ln \left( \frac{1}{1 - \alpha} \right)$
double hashing mit unabhängigen $h$ u. $h'$	$\frac{1}{\alpha} \ln \left( \frac{1}{1 - \alpha} \right)$	$\frac{1}{1 - \alpha}$

- Die Angaben setzen eine ideale Hashfunktion voraus.  
D.h. die Schlüssel sind gleichmäßig über die Tabelle verstreut.
- Ergebnisse aus [Ottmann u. Widmayer 2002] und [Cormen et. al, 2010].

# Komplexität der Hashverfahren für konkrete Belegungsgrade

	$\alpha = 0.5$		$\alpha = 2/3$		$\alpha = 0.8$	
Verfahren	$C_{\text{Erfolg}}$	$C_{\text{Nicht-Erfolg}}$	$C_{\text{Erfolg}}$	$C_{\text{Nicht-Erfolg}}$	$C_{\text{Erfolg}}$	$C_{\text{Nicht-Erfolg}}$
Hashverfahren mit Verkettung	1.25	0.5	1.33	0.66	1.4	0.8
Offenes Hashverfahren mit linearem Sondieren	1.5	2.5	2	5	3	13
Offenes Hashverfahren mit alternierend quadratischem Sondieren	1.44	2.19	1.77	3.43	2.21	5.81
double hashing mit unabhängigen $h$ u. $h'$	1.38	2	1.65	3	2.01	5

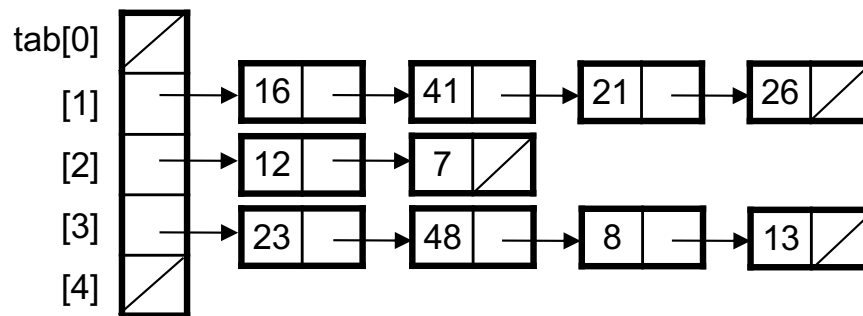
- Bei offenen Hashverfahren wird in der Praxis üblicherweise ein Füllungsgrad von  $\alpha \leq 2/3$  angestrebt.

## 2. Suchen mit Hashverfahren

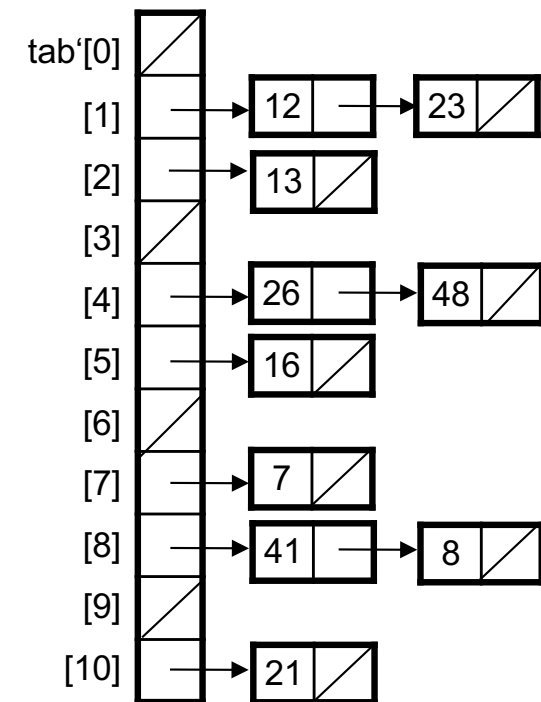
- Idee
- Hashfunktion
- Hashverfahren mit linear verketteten Listen
- Offene Hashverfahren
- **Dynamische Hashverfahren**
- Hashverfahren in Java

# Vergrößern der Tabelle mit sofortigem Umkopieren

- Falls bestimmter Füllungsgrad  $\alpha$  überschritten:
  - Lege neue Tabelle  $\text{tab}'$  mit einer in etwa doppelten Größe  $m'$  ( $m'$  Primzahl).
  - Füge alle Einträge aus alter Tabelle  $\text{tab}$  in die neue Tabelle  $\text{tab}'$  ein.
- Umorganisation der Hashtabelle ist bei großen Tabellen singulär sehr aufwendig.



$$\alpha = 10/5 = 200\%$$



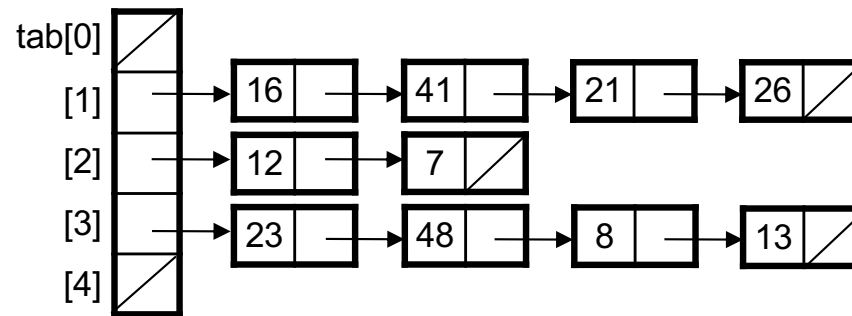
$$\alpha = 10/11 = 90\%$$

# Vergrößern der Tabelle mit verzögertem Umkopieren (lazy copying)

---

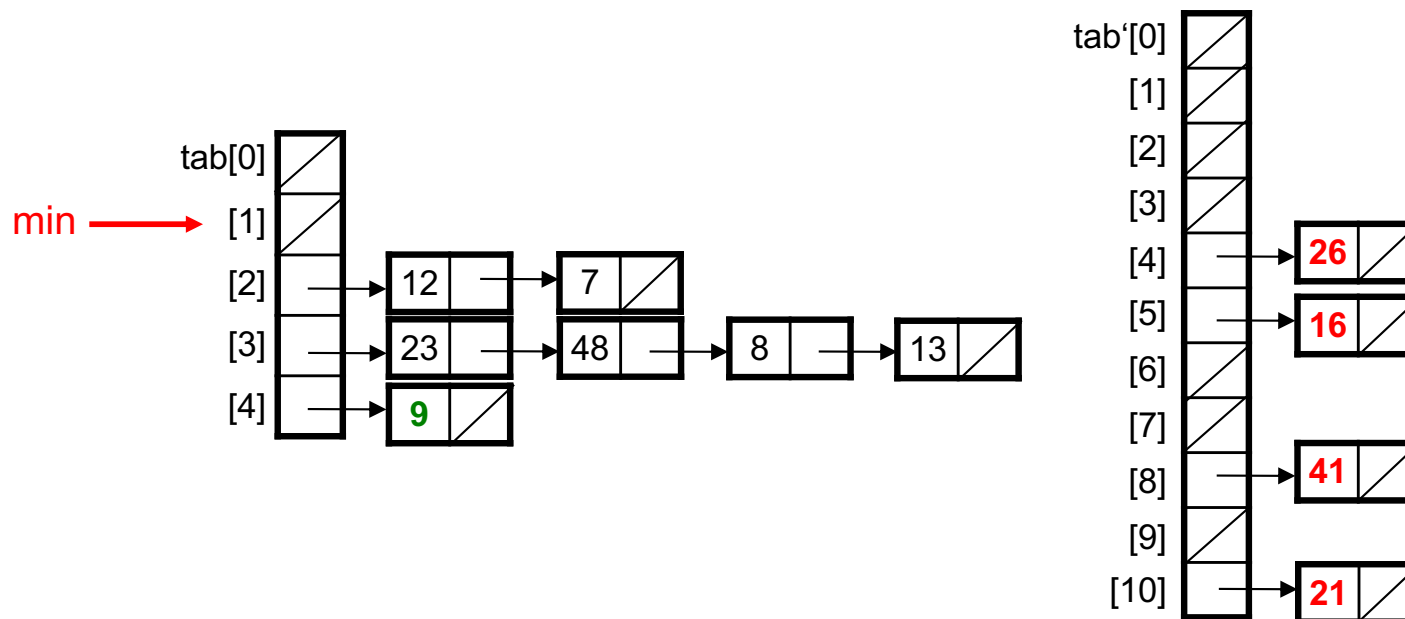
- Nur für Hashverfahren mit Verkettung.
- Falls bestimmter Füllungsgrad überschritten wird:
  - Lege neue Tabelle  $tab'$  mit einer in etwa doppelten Größe  $m'$  an ( $m'$  sollte Primzahl sein).
  - Bei jedem Zugriff auf die Tabelle (search, insert bzw. remove) wird zusätzlich der nicht-leere Tabelleneintrag (komplette verkettete Liste) mit kleinstem Index  $min$  in die neue Tabelle übertragen.
  - Entscheide bei jedem Zugriff, ob Daten in alter oder neuer Tabelle stehen: Falls  $h(k) = k \bmod m \leq min$ , dann greife auf neue Tabelle  $tab'$  zu, sonst auf alte Tabelle  $tab$ .
  - Beachte, dass bei alter Tabelle  $tab$  mit  $h(k) = k \bmod m$  und bei neuer Tabelle  $tab'$  mit  $h'(k) = k \bmod m'$  zugegriffen wird.
  - Falls alte Tabelle  $tab$  nicht mehr gebraucht wird (d.h.  $min = \text{Größe der Tabelle } tab$ ), wird  $tab$  durch  $tab'$  ersetzt.

# Beispiel zu Vergrößern der Tabelle mit verzögertem Umkopieren (1)



insert(9):

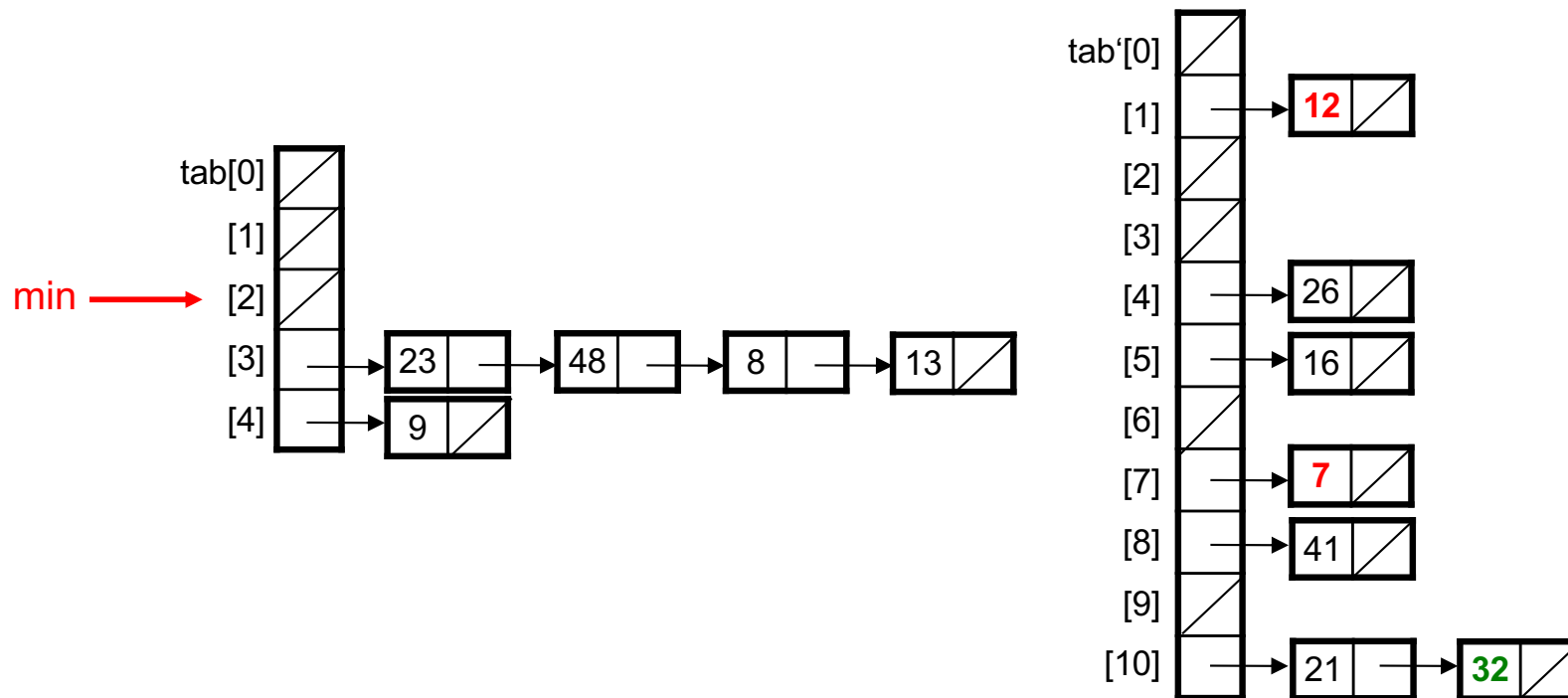
- Lege neue Tabelle tab' der Größe 11 an, da Füllungsgrad von tab überschritten wird.
- Übertrage kleinsten Tabelleneintrag tab[min] mit  $\min = 1$  nach tab'.
- Füge nun 9 in alte Tabelle ein (da  $h(9) = 9 \bmod 5 = 4 > \min$ ).



## Beispiel zu Vergrößern der Tabelle mit verzögertem Umkopieren (2)

insert(32):

- Übertrage kleinsten Tabelleneintrag  $\text{tab}[\text{min}]$  mit  $\text{min} = 2$  nach  $\text{tab}'$ .
- Füge nun 32 in neue Tabelle ein ( $h(32) = 32 \bmod 5 = 2 \leq \text{min}$ ).  
Da  $h'(32) = 32 \bmod 11 = 10$  ist, wird 32 beim Index 10 eingetragen.

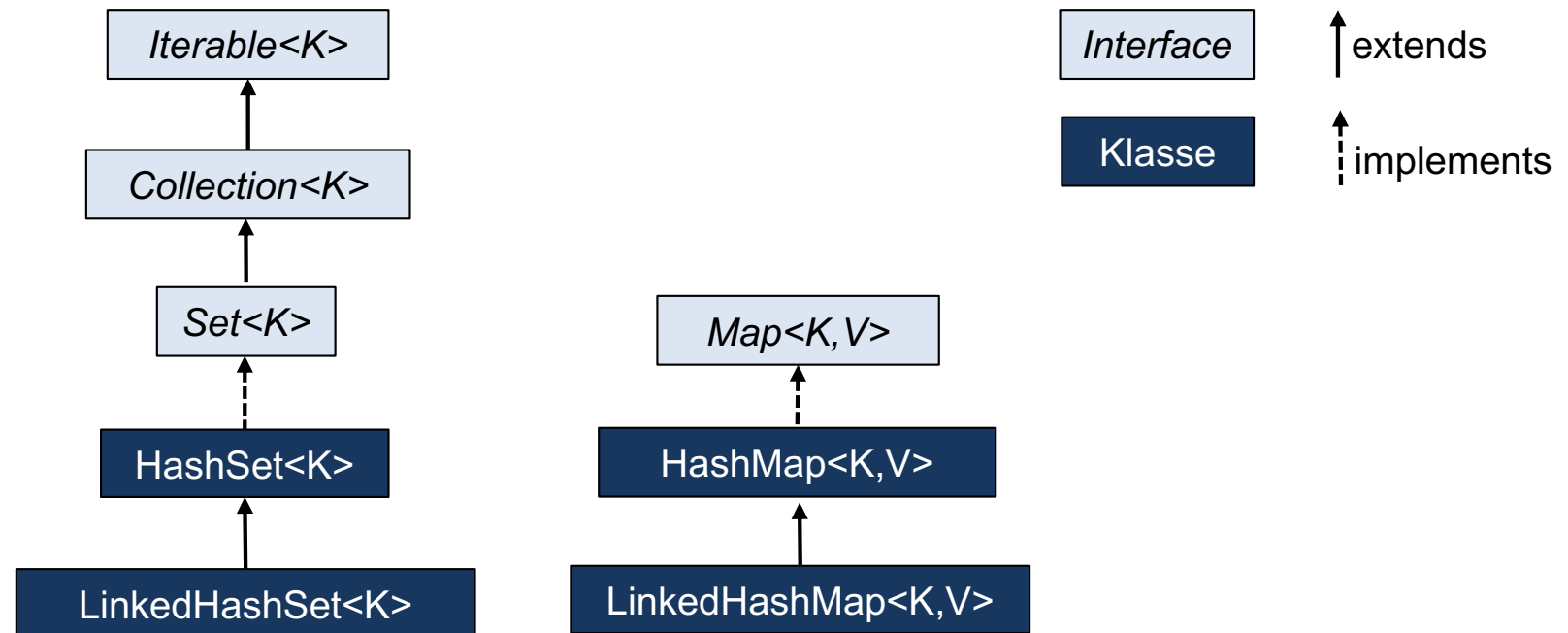




## 2. Suchen mit Hashverfahren

- Idee
- Hashfunktion
- Hashverfahren mit linear verketteten Listen
- Offene Hashverfahren
- Dynamische Hashverfahren
- Hashverfahren in Java

# Hashverfahren in Java



- **HashSet** und **HashMap** verwenden **linear verkettete Listen für kollidierende Schlüssel**
- Bei **LinkedHashSet** und **LinkedHashMap** wird zusätzlich mit einer doppelt verketteten Liste über die Reihenfolge des Einfügens Buch geführt.
- Bei **LinkedHashMap** kann statt der Einfüge-Reihenfolge auch die Zugriffs-Reihenfolge eingestellt werden: **least-recently to most-recently accessed**.
- Wichtig: **hashCode** und **equals** muss für den Schlüsseltyp geeignet überschrieben werden. Falls `o1.equals(o2)`, dann muss auch `o1.hashCode() == o2.hashCode()` sein.
- Außer **HashMap** gibt es noch die Varianten **WeakHashMap** und **IdentityHashMap**. Sind als **offene Hashverfahren mit linearer Sondierung** realisiert.