

# Hashing und Hashmaps



**Dozent:** Prof. Dr. Michael Eichberg  
**Kontakt:** michael.eichberg@dhw.de, Raum 149B  
**Version:** 2.0.1  
**Quelle:** Die Folien sind teilweise inspiriert von oder basierend auf Robert Sedgewick und Kevin Wayne, "Algorithms", Addison-Wesley, 4th Edition, 2011 sowie auf Lehrmaterial von Prof. Dr. Ritterbusch

---

**Folien:** <https://delors.github.io/theo-algo-hashing/folien.de.rst.html>  
<https://delors.github.io/theo-algo-hashing/folien.de.rst.html.pdf>

**Fehler melden:** <https://github.com/Delors/delors.github.io/issues>

# 1. Einführung

## Suchen in einer Liste

Implementation	Garantie			Durchschnittlicher Fall			Operati auf d Schlüss
	Suchen	Einfügen	Löschen	Suchen	Einfügen	Löschen	
sequentielle Suche (unsortierte Liste)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$	equals()
binäre Suche (geordnetes Array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	compar
BST [1]	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	compar

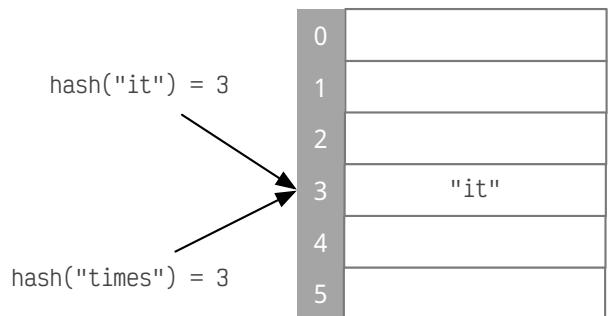
? Frage

Können wir effizienter suchen?

[1] Binary Search Tree

# Hashing - Grundidee

- Die Elemente werden über den Schlüssel indexiert in einer Tabelle gespeichert.  
Der Index ist eine Funktion des Schlüssels.
- Hash-Funktion: Methode zur Berechnung des Array-Index aus dem Schlüssel.



## Herausforderungen

1. Berechnung der Hash-Funktion.
2. Gleichheitstest: Methode zur Überprüfung, ob zwei Schlüssel gleich sind.
3. Kollisionsauflösung: Algorithmus und Datenstruktur zur Behandlung von zwei Schlüsseln, die auf denselben Array-Index hindeuten.

### ※ Hinweis

Klassischer Kompromiss zwischen Raum und Zeit!

- Keine Platzbeschränkung: triviale Hash-Funktion mit Schlüssel als Index.
- Keine Zeitbeschränkung: triviale Kollisionsauflösung mit sequentieller Suche.
- Raum- und Zeitbeschränkung: Hashing (die reale Welt).

In dem Beispiel ist der Schlüssel das Wort `it`.

# Berechnung der Hash-Funktion

**Idealistisches Ziel:** Die Schlüssel gleichmäßig verwürfeln, um einen Tabellenindex zu erzeugen.

- Effizient berechenbar.
  - Jeder Tabellenindex ist für jeden Schlüssel gleich wahrscheinlich.
- 

Die Frage, wie man gute Schlüssel berechnet, ist ein gründlich erforschtes Problem, dass in der Praxis immer noch problematisch ist.

## Beispiel 1. Telefonnummern.

Schlecht: die ersten drei bis fünf Ziffern.

Besser: die letzten vier Ziffern.

## Beispiel 2. Sozialversicherungsnummer

Schlecht: die ersten beiden Ziffern.

Besser: die letzten Ziffern.

---

Die ersten beiden Stellen bei der Sozialversicherungsnummer identifizieren den Rentenversicherungsträger.

Praktische Herausforderung: für jeden Schlüsseltyp ist ein anderer Ansatz erforderlich.

# Hashfunktionen

## Definition

Eine Hashfunktion  $h : M \rightarrow \mathbb{Z}_n$  bildet eine Menge  $M$  mit  $|M| \geq |\mathbb{Z}_n|$  auf die Zahlen  $0, \dots, n - 1$  ab.

Eine Hashfunktion ist *surjektiv* [2]: für jedes  $y \in \mathbb{Z}_n$  gibt es ein  $x \in M$  mit  $h(x) = y$ .

Eine Hashfunktion ist *gleichverteilt*, wenn zwei Bilder  $y_1, y_2 \in \mathbb{Z}_n$  immer ungefähr gleich viele Urbilder haben  $|h^{-1}(y_1)| \approx |h^{-1}(y_2)|$ .

## Hashes für unterschiedliche Anwendungen

- **Hashes für Datenstrukturen** müssen sehr effizient sein.
- Für **Hashes, welche verwendet werden im Rahmen von Verschlüsselung und Signaturen**, muss es schwer sein:

- ein Urbild zu finden (d. h. von Y auf X zu schließen)
- zwei kollidierende Werte zu finden.

---

MD5 ist seit 2008 und SHA1 seit 2017 „geknackt“.

- kryptographische Hashes sollten effizient berechenbar sein.

- **Hashes für Passwortspeicherung** müssen die selben Anf. erfüllen wie Hashes für Signaturen und Verschlüsselungszwecke, dürfen aber nicht effizient berechenbar sein.

## !! Wichtig

Im Folgenden konzentrieren wir uns auf Hashes für Datenstrukturen.

Wenn das Ziel ist, Hash-Werte mit einer bestimmten Länge (zum Beispiel 32Bit) zu berechnen, dann wären folgende Hashfunktionen denkbar:

## Exemplarische Hashfunktionen

### Ganze Zahlen

```
1 | hash(x: u32) : u32 { return x; } // u32 = 32-Bit unsigned integer
```

### Gleitkommazahlen

```
1 | hash(x: f64) : u32 { // f64 = 64-Bit (IEEE) floating point number
2 |   bits : u64 = f64ToBits(x); // u64 = 64-Bit (signed) integer
3 |   return (u32)(bits ^ (bits >>> 32));
4 }
```

---

>>> ist der *unsigned right shift* Operator.

## Zeichenketten

## Horners Methode für Zeichenketten der Länge L:

$$h = s[0] \cdot 31^{L-1} + \dots + s[L-2] \cdot 31^1 + s[L-1] \cdot 31^0.$$

```
1 fn hash(s: [char,4]) : u32 {  
2     hash: u32 = 0  
3     for i in 0..4 { hash = s[i] + hash * 31; }  
4     return hash;  
5 }  
6 // hash(['c','a','l','l']) = // ≈ hash("call")  
7 //      99 · 31 · 31 + 97 · 31 · 31 + 108 · 31 + 108 =  
8 //      108 + 31 · ( 108 + 31 · (97 + 31 · (99)))
```

Bemerkung	
char	unicode
'a'	97
'b'	98
'c'	99
:	:
'l'	108

- [2] In machen Fällen ist der Nachweis nicht möglich, aber es wird vermutet.

## 2. Hashing in Python

# Verwendung von Hashes in Python

- Bei der Speicherung von Objekten in Sets und Dictionaries verwendet Python Hashes.
- Sobald ein Objekt in einem Set oder Dictionary gespeichert wird, darf der Objektzustand (zumindest im Hinblick auf die Hashfunktion) nicht mehr verändert werden!
- Der Hashwert eines (nicht veränderlichen) Objekts kann mit der Funktion `hash()` berechnet werden.
- Eigene Objekte in Sets und Dictionaries speichern:
  - Um benutzerdefinierte Objekte in einer Hashmap zu speichern, müssen wir die Methoden `_hash_` und `_eq_` implementieren.
  - Zu beachten:
    - Hashwerte *müssen für gleiche Objekte gleich sein.*
    - Hashwerte *für unterschiedliche Objekte sollten unterschiedlich sein.*

## Beispielklasse Person

```
1 class Person:  
2     def __init__(self, name, age):  
3         self.name = name  
4         self.age = age  
5  
6     def __eq__(self, other):  
7         if isinstance(other, Person):  
8             return self.name == other.name and \  
9                     self.age == other.age  
10        return False  
11  
12    def __hash__(self):  
13        return hash((self.name, self.age))
```

## Verwendung der Klasse Person

```
1 person1 = Person("Alice", 30)  
2 person2 = Person("Bob", 25)  
3 person3 = Person("Alice", 30) # gleiche Werte wie "person1"
```

## Beispielausgabe

```
>>> person1  
<__main__.Person object at 0x101474c20>  
>>> person2  
<__main__.Person object at 0x1013daad0>  
>>> person3  
<__main__.Person object at 0x1013db110>
```

## Speicherung von Person-Objekten in einem Set

```
1 | people = {person1, person2, person3}
```

## Ausgabe des Sets

```
1 | for p in people: print(p.name)
```

## Ausgabe

Bob  
Alice

## Verwendung der hash-Funktion

```
1 | print(hash(person1))  
2 | print(hash(person2))  
3 | print(hash(person3))
```

## Beispielausgabe

3529483511948588452  
-9048922068811934735  
3529483511948588452

---

In Python ist die Ausgabe der Funktion `hash()` nach jedem Neustart der Pythonumgebung unterschiedlich, da die Hashfunktion einen Zufallswert enthält, der bei jedem Neustart neu generiert wird.

## Beispielklasse Person mit konstantem Hashwert

```
1 | class PersonWithBadHash:  
2 |     def __init__(self, name, age):  
3 |         self.name = name  
4 |         self.age = age  
5 |  
6 |     def __eq__(self, other):  
7 |         if isinstance(other, Person):  
8 |             return self.name == other.name and \  
9 |                     self.age == other.age  
10 |         return False  
11 |  
12 |     def __hash__(self):  
13 |         return 1 # immer der gleiche Hashwert
```

---

Die Verwendung des Alters der Person als Hashwert wäre in den allermeisten Fällen auch keine gute Idee, da es (vermutlich) viele Hashkollisionen geben würde.

## Verwendung einer Klasse mit einer konstanten Hashfunktion

```
person1 = Person("Alice", 30)  
person2 = Person("Bob", 25)  
person3 = Person("Alice", 30)  
people = {person1, person2, person3}  
print(hash(person1))  
print(hash(person2))  
print(hash(person3))  
print(" ".join(map(lambda p: p.name, people)))
```

## Beispielausgabe

1  
1  
1  
Alice Bob

### ⚠ Warnung

Die Verwendung einer konstanten Hashfunktion ist in der Regel keine gute Idee,  
da sie die Effizienz von Hashmaps ganz erheblich beeinträchtigen kann.

# Übung

## 2.1. Eine Klasse zur Repräsentation von Studierenden.

Die Klasse `Student` soll:

- die Attribute/Properties `name` und `matriculation_number` haben.
- die Methoden `__eq__` und `__hash__` sinnvoll/korrekt definieren

Aufgaben:

1. Erzeugen Sie drei `Student`-Objekte und speichern Sie diese in einem Set.
2. *Fragen Sie sich wie sie effizient den Hashwert berechnen können.*
3. Geben Sie die Namen der Studierenden aus.
4. Was passiert, wenn Sie — *nachdem Sie ein Student Objekt dem Dictionary hinzugefügt haben* — den Namen des Studenten ändern?

Schreiben Sie entsprechenden Code, um Ihre Annahme zu überprüfen!

---

### Rumpfimplementierung

```
1 class Student:  
2     def __init__(self, ...):  
3         raise NotImplementedError("TODO")  
4  
5     def __eq__(self, other):  
6         raise NotImplementedError("TODO")  
7  
8     def __hash__(self):  
9         raise NotImplementedError("TODO")
```

### 3. Hashing in Java

# Verwendung von Hashes in Java

- Bei der Speicherung von Objekten in Sets und `HashMaps/Hashtables` verwendet Java Hashes.
- Sobald ein Objekt in einem Set oder einer Map gespeichert wird, darf der Objektzustand (zumindest im Hinblick auf die Hashfunktion) nicht mehr verändert werden!
- Der Hashwert eines (nicht veränderlichen) Objekts kann mit der Funktion `hashCode()` berechnet werden.
- Eigene Objekte in Sets und Maps speichern:
  - Um benutzerdefinierte Objekte in einer `HashMap` zu speichern, müssen wir die Methoden `boolean equals(Object o)` und `int hashCode()` implementieren.
  - Zu beachten:
    - Hashwerte müssen für gleiche Objekte gleich sein.
    - Hashwerte für unterschiedliche Objekte sollten unterschiedlich sein.

## Beispielklasse Person

```
1 class Person {  
2     private String name;  
3     private int age;  
4     Person(String name, int age) { this.name = name; this.age = age; }  
5  
6     public boolean equals(Object o) {  
7         if (o instanceof Person) { // Alt. compare class objects  
8             Person other = (Person) o;  
9             return this.name.equals(other.name) && this.age == other.age;  
10        }  
11        return false;  
12    }  
13  
14    public int hashCode() { return java.util.Objects.hash(name, age); }  
15 }
```

## Verwendung der Klasse Person

```
1 var person1 = new Person("Alice", 30);  
2 var person2 = new Person("Bob", 25);  
3 var person3 = new Person("Alice", 30); // gleiche Werte wie "person1"
```

## Beispielausgabe

```
person1 => Person@750e297f // the addresses are memory addresses  
person2 => Person@1fb0e5  
person3 => Person@650e893b
```

## Speicherung von Person-Objekten in einem Set

```
1 var set = new HashSet<Person>();  
2 set.add(person1);  
3 set.add(person2);
```

```
4 | set.add(person3);  
1 | // throws IllegalArgumentException:  
2 | var people = Set.of(person1, person2, person3)
```

## Ausgabe des Sets

```
1 | for (var p : people) System.out.println(p.name);
```

## Ausgabe

```
Bob  
Alice
```

## Verwendung der hashCode-Funktion

```
1 | System.out.println(person1.hashCode());  
2 | System.out.println(person2.hashCode());  
3 | System.out.println(person3.hashCode());
```

## Beispieldausgabe

```
1963862399  
2076901  
1963862399
```

## Beispielklasse Person mit konstantem Hashwert

```
1 | class PersonWithBadHash {  
2 |     String name;  
3 |     int age;  
4 |     PersonWithBadHash(String name, int age) { this.name = name; this.age = age; }  
5 |  
6 |     public boolean equals(Object o) {  
7 |         if (o instanceof PersonWithBadHash) {  
8 |             PersonWithBadHash other = (PersonWithBadHash) o;  
9 |             return this.name.equals(other.name) && this.age == other.age;  
10 |        }  
11 |        return false;  
12 |    }  
13 |  
14 |    public int hashCode() { return 1; /* immer der gleiche Hashwert */ }  
15 | }
```

---

Die Verwendung „nur“ des Alters der Person als Hashwert wäre in den allermeisten Fällen auch keine gute Idee, da es (vermutlich) viele Hashkollisionen geben würde.

## Verwendung einer Klasse mit einer konstanten Hashfunktion

```
var person1 = new PersonWithBadHash("Alice", 30);  
var person2 = new PersonWithBadHash("Bob", 25);  
System.out.println(person1.hashCode());  
System.out.println(person2.hashCode());  
var people = Set.of(person1, person2);  
people.forEach(p → System.out.println(p.name));
```

## Beispielausgabe

1  
1  
1  
Alice Bob

### ⚠ Warnung

Die Verwendung einer konstanten Hashfunktion ist in der Regel keine gute Idee,  
da sie die Effizienz von Hashmaps ganz erheblich beeinträchtigen kann.

# Übung

## 3.1. Eine Klasse zur Repräsentation von Studierenden.

Die Klasse `Student` (Nutzen Sie hier kein `record`) soll:

- die Attribute/Properties `name` und `matriculationNumber` haben.
- die Methoden `equals` und `hashCode` sinnvoll/korrekt definieren

Aufgaben:

1. Erzeugen Sie drei `Student`-Objekte und speichern Sie diese in einem Set.
2. *Fragen Sie sich wie sie effizient den Hashwert berechnen können.*
3. Geben Sie die Namen der Studierenden aus.
4. Was passiert, wenn Sie — *nachdem Sie ein Student Objekt einem (Hash)Set hinzugefügt haben* — den Namen des Studenten ändern?

Schreiben Sie entsprechenden Code, um Ihre Annahme zu überprüfen!

---

## Rumpfimplementierung

```
1 import java.util.HashSet;
2
3 class Student {
4     private String name;
5     private int matriculationNumber;
6
7     public Student(String name, int matriculationNumber) {
8         this.name = name;
9         this.matriculationNumber = matriculationNumber;
10    }
11
12    public String getName() { return name; }
13    public void setName(String name) { this.name = name; }
14    public int getMatriculationNumber(){ return matriculationNumber; }
15    public void setMatriculationNumber(int matriculationNumber) {
16        this.matriculationNumber = matriculationNumber;
17    }
18
19    @Override public boolean equals(Object o) {
20        throw new UnsupportedOperationException("TODO");
21    }
22
23    @Override public int hashCode() {
24        throw new UnsupportedOperationException("TODO");
25    }
26 }
```



## 4. Hashfunktionen

# Gängige Ansätze für Hashfunktionen

## Modulo-Hashfunktion:

Sei  $n$  möglichst eine Primzahl:

$$h_n^{mod}(x) = x \bmod n$$

### Bewertung

- einfach zu berechnen/sehr effizient
- surjektiv
- gleichverteilt
- wenn  $n$  keine Primzahl ist, dann kann es (leicht) passieren, dass bestimmte (Teil-)daten weniger oder keinen Einfluss auf den Hashwert haben:
  - $x \cdot 10^3 \bmod 40 = 0$
  - $x \cdot 10^3 \bmod 42 \in \{0, 2, 4, \dots, 40\}$  Anm.:  $\text{ggT}(42, 1000) = 2$
  - $x \cdot 10^3 \bmod 41 \in \{0, 1, 2, 3, \dots, 40\}$  Anm.:  $\text{ggT}(41, 1000) = 1$

## Multiplikations-Hashfunktion:

Sei  $c$  fest, oft  $c = \frac{\sqrt{5}-1}{2} \approx 0,6180339887498949$ :

$$h_n^{mul}(x) = \lfloor n \cdot (c \cdot x - \lfloor c \cdot x \rfloor) \rfloor$$

### Bewertung

- nicht beweisbar surjektiv
- nur asymptotisch gleichverteilt
- Das verwendete  $c$  sollte eine gute Durchmischung der Key-Bits fördern.  
Andere irrationale Zahlen sind ggf. auch sinnvoll/möglich.
- Berechnung benötigt eine effiziente Fließkomma-Verarbeitung

# Übung

## 4.1. Hashwerte berechnen I

Berechnen Sie:

1.  $h_{257}^{mod}(1\ 000)$
2.  $h_{257}^{mul}(1\ 000)$

## 4.2. Hashwerte berechnen II

Berechnen Sie:

1.  $h_{263}^{mod}(10\ 000)$
2.  $h_{263}^{mul}(10\ 000)$

## 5. Hashtabellen ( *Hashmaps* oder *Dictionaries*)

# Grundlagen von Hashtabellen

Das Grundprinzip von Hashtabellen ist einfach:

- Im Vorfeld wird ein Array  $A$  einer Größe  $n$  angelegt,  
Die Größe des Arrays übersteigt die erwartete Belegung deutlich.
- Daten mit einem Schlüssel  $k$  werden dann an der Position  $A[h(k)]$  gespeichert  
- oder an einer Ersatzposition.
- Sollte die Belegung zu groß werden, wird das Array vergrößert und die Elemente  
werden (ggf.) neu bzw. wieder gehasht.
- Sollten zwei Schlüssel den gleichen Hash haben (d. h.  $h(x_1) = h(x_2)$ ), dann wird  
eine Kollisionsauflösung benötigt.

# Belegung von Hashtabellen

Die Belegung von Hashtabellen ist für die Effizienz entscheidend.

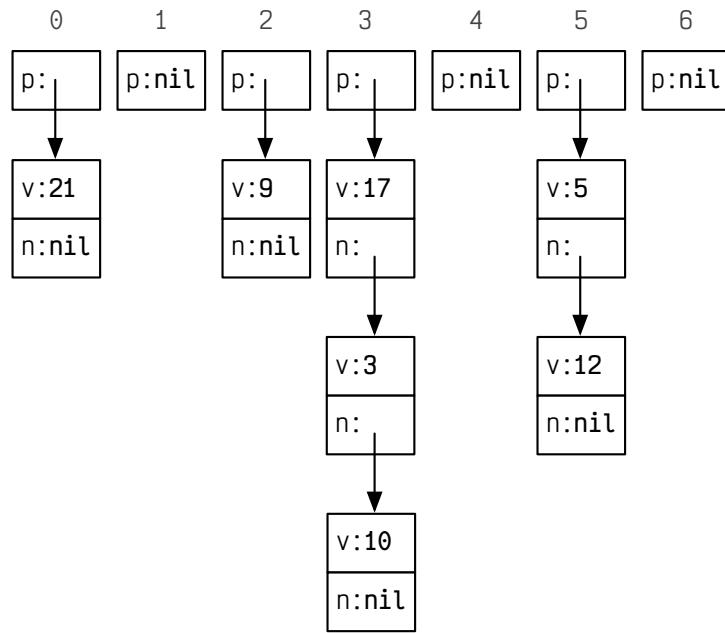
## Definition

Ein Array  $A$  der Kapazität  $n$  mit einer Hashfunktion  $h_n$  wird  
*Hashtabelle*( $A, h_n$ ) genannt.

Sind zu einem Zeitpunkt  $m$  (erste) Felder belegt, so hat die *Hashtabelle*( $A, h_n$ )  
eine Belegung von  $\alpha = \frac{m}{n}$ .

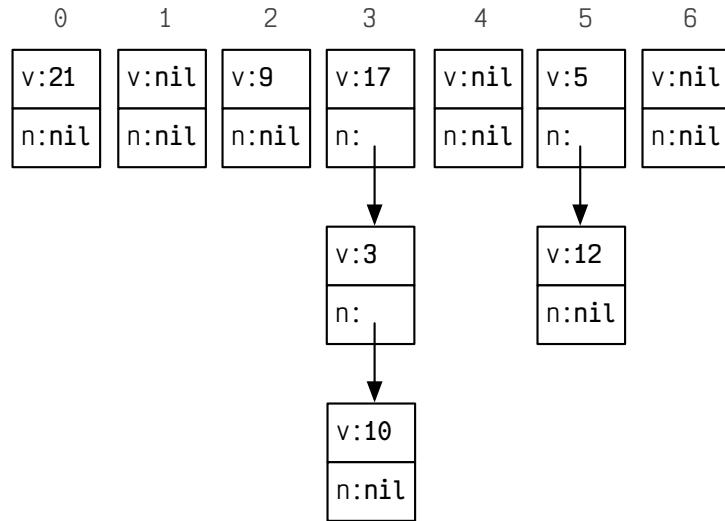
# Verkettete Hashtabellen

## Direkte Verkettung



Die *direkte Verkettung* von Überläufern verwendet eine *Hashtabelle* ( $A, h_n$ ), mit einem Array  $A$ , das aus Zeigern auf einfach verkettete Listen besteht, dessen Schlüssel der Einträge alle den gleichen Hashwert besitzen, oder die `nil` sind, wenn kein Eintrag bisher mit dem jeweiligen Hashwert vorhanden ist.

## Separate Verkettung



Die *separate Verkettung* von Überläufern verwendet eine *Hashtabelle* ( $A, h_n$ ), bei der das Array  $A$  aus Knoten einer einfach verketteten Liste besteht, dessen Wert `nil` ist, wenn unter dem Hashwert noch nichts gespeichert wurde.

Ein Eintrag mit Schlüssel  $k$  wird der verketteten Liste zugeordnet, die in  $A[h_n(k)]$  verlinkt ist oder startet, und kann entsprechend hinzugefügt, gelöscht und gefunden werden.



# Offene Adressierung

## Definition

Soll der *Hashtabelle* ( $A, h_n$ ) mit einem Array  $A$  ein Datensatz mit Schlüssel  $k$  hinzugefügt werden soll, so erfolgt dies in  $A[h_n(k)]$ , wenn dieser Eintrag noch nicht belegt ist. Ansonsten werden  $i = 1, \dots, n - 1$  weitere Positionen  $A[g_n(k, i)]$  geprüft.

## Strategien

### Lineares-Sondieren:

Das Array wird linear durchsucht.

$$g_n^{lin}(k, i) = (h_n(k) + i) \bmod n$$

### Quadratisches-Sondieren:

Das Array wird quadratisch steigend durchsucht.

$$g_n^{quad}(k, i) = (h_n(k) + i^2) \bmod n$$

### Doppeltes-Hashing:

Das Array wird mit Hilfe einer zweiten Hashfunktion:

$$h'_n(k) = (k \bmod (n - 2)) + 1$$

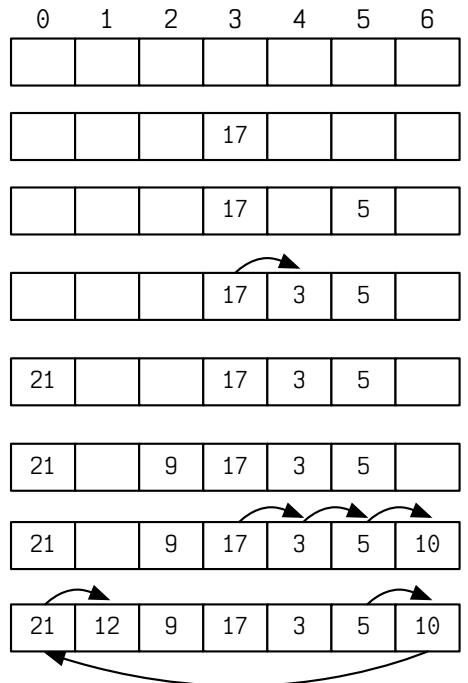
durchsucht.

$$g_n^{doppel}(k, i) = (h_n(k) + i \cdot h'_n(k)) \bmod n$$

# Beispiel Offene Adressierung (Hashing: $x \bmod 7$ )

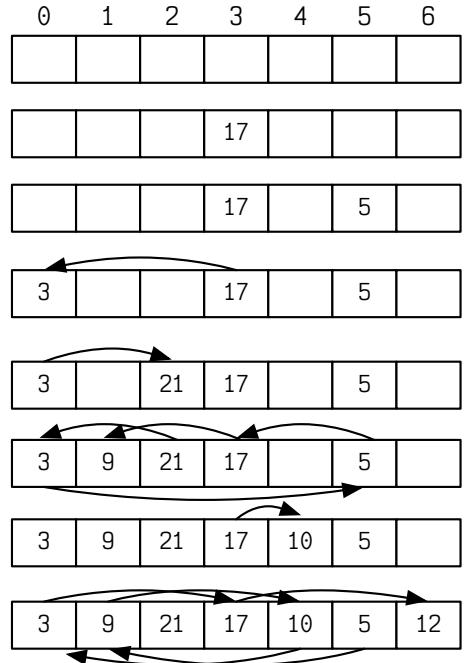
## Lineare Sondierung

Hinzufügen von (17, 5, 3, 21, 9, 10, 12)



## Doppeltes-Hashing

Hinzufügen von (17, 5, 3, 21, 9, 10, 12)



## Quadratische Sondierung

Hinzufügen von (17, 5, 3, 21, 9, 10, 12)

Für den Wert 10 wird kein Platz gefunden!

$$(10 \bmod 7 = 3)$$

1.  $3 + 0^2 \bmod 7 = 3$
2.  $3 + 1^2 \bmod 7 = 4$
3.  $3 + 2^2 \bmod 7 = 0$
4.  $3 + 3^2 \bmod 7 = 5$
5.  $3 + 4^2 \bmod 7 = 5$

6.  $3 + 5^2 \bmod 7 = 0$

7.  $3 + 6^2 \bmod 7 = 4$

0	1	2	3	4	5	6

			17			
--	--	--	----	--	--	--

			17		5	
--	--	--	----	--	---	--

			17	3	5	
--	--	--	----	---	---	--

21			17	3	5	
----	--	--	----	---	---	--

21		9	17	3	5	
----	--	---	----	---	---	--

21		9	17	3	5	
----	--	---	----	---	---	--

21		9	17	3	5	12
----	--	---	----	---	---	----

# Übung

## 5.1. Werte in kleine Hashtabelle einfügen

Belegen Sie eine Hashtabelle mit  $n = 5$  Feldern mit den Werten 37, 18, 32 und 24 auf Basis von  $h_5^{mod}(x)$  mit linearer Sondierung, quadratischer Sondierung und doppeltem Hashing mit  $h_5'(x) = (x \bmod 3) + 1$ .

## 5.2. Werte in größere Hashtabelle einfügen

Belegen Sie eine Hashtabelle mit  $n = 11$  Feldern mit den Werten 37, 49, 26 und 39 auf Basis von  $h_{11}^{mod}(x)$  mit linearer Sondierung, quadratischer Sondierung und doppeltem Hashing mit  $h'_{11}(x) = (x \bmod 9) + 1$ .

## Angriffe auf algorithmische Komplexität

Julian Wälde and Alexander Klink reported that the `String.hashCode()` hash function is not sufficiently collision resistant.

`hashCode()` value is used in the implementations of [Java 6] HashMap and Hashtable classes. A specially-crafted set of keys could trigger hash function collisions, which can degrade performance of HashMap or Hashtable by changing hash table operations complexity from an expected/average  $O(1)$  to the worst case  $O(n)$ . Reporters were able to find colliding strings efficiently using equivalent substrings and meet in the middle techniques. This problem can be used to start a denial of service attack against applications that use untrusted inputs as HashMap or Hashtable keys. An example is a web application server that may fill hash tables with data from HTTP request. A remote attack could use that to make JVM use excessive amount of CPU time by sending a POST request with large amount of parameters which hash to the same value.

—[Abbreviated Version] Jan Lieskovsky 2011-11-01