

# Algorithmen und Datenstrukturen (Th. Ottmann und P. Widmayer)

**Folien: Hashverfahren**

**Autor: Sven Schuierer & Wolfram Burgard**

Institut für Informatik  
Georges-Köhler-Allee  
Albert-Ludwigs-Universität Freiburg

# Hashverfahren

## Wörterbuchproblem:

Suchen, Einfügen, Entfernen von Datensätzen  
(Schlüsseln)

Ort des Datensatzes  $d$ : Berechnung aus dem Schlüssel

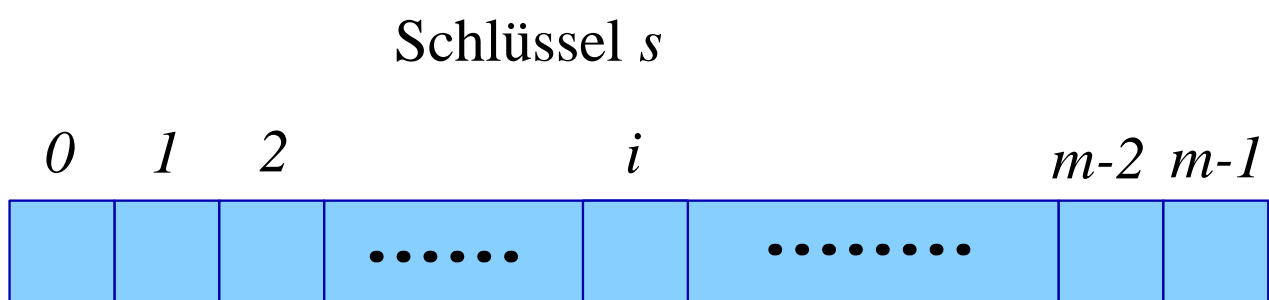
$s$  von  $d$

⇒ keine Vergleiche

⇒ konstante Zeit

Datenstruktur: lineares Feld (Array) der Größe  $m$

Hashtabelle



# Hashverfahren—Beispiele

## Beispiele:

- Compiler

i	int	0x87C50FA4
j	int	0x87C50FA8
x	double	0x87C50FAC
name	String	0x87C50FB2
:		

- Umgebungsvariablen ([Schlüssel](#), [Attribut](#)) Liste

```
EDITOR=emacs  
GROUP=mitarbeiter  
HOST=vulcano  
HOSTTYPE=sun4  
LPDEST=hp5  
MACHTYPE=sparc  
:
```

- ausführbare Programme

```
PATH=~/.bin:/usr/local/gnu/bin:  
/usr/local/bin:/usr/bin:/bin: . . .
```

# Hashverfahren—Probleme

## 1. Größe der Hashtabelle

Nur eine kleine Teilmenge  $S$  aller möglichen Schlüssel (des **Universums**)  $\mathcal{U}$  kommen vor

## 2. Berechnung der Adresse eines Datensatzes

- Schlüssel sind keine ganzen Zahlen
- Index hängt von der Größe der Hashtabelle ab

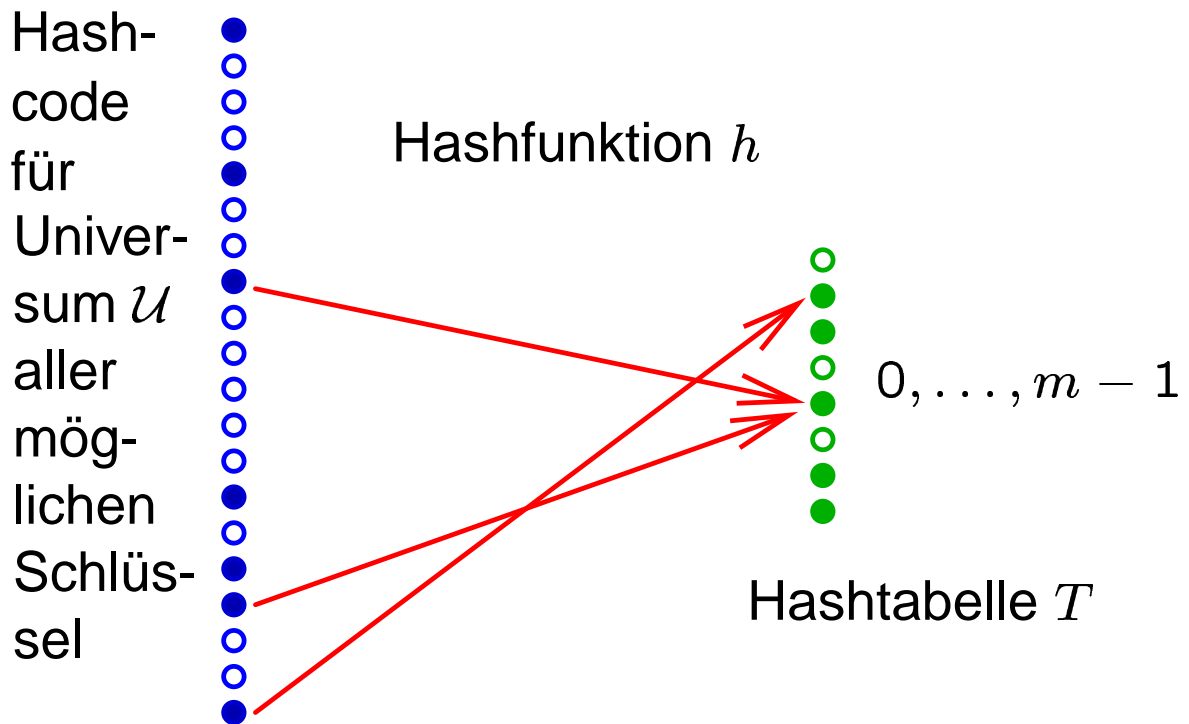
In Java:

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

Das Universum  $\mathcal{U}$  sollte möglichst **gleichmäßig** auf die Zahlen  $-2^{31}, \dots, 2^{31} - 1$  verteilt werden

# Hashfunktion

Schlüsselmenge  $S$



$$(H(\mathcal{U}) \subseteq [-2^{31}, 2^{31} - 1])$$

$h(s) =$  **Hashadresse**

$h(s) = h(s') \iff s$  und  $s'$  sind **Synonyme** bzgl.  $h$   
**Adresskollision**

# Hashverfahren

**Beispiel** für  $\mathcal{U}$ : alle Namen in Java mit Länge  $\leq 40$   
 $\Rightarrow |\mathcal{U}| = 62^{40}$

Falls  $|\mathcal{U}| > m$ : Adresskollisionen unvermeidlich

## Hashverfahren:

1. Wahl einer möglichst „guten“ Hash-Funktion
2. Strategie zur Auflösung von Adresskollisionen

## Belegungsfaktor $\alpha$

$$\alpha = \frac{\# \text{ gespeicherte Schlüssel}}{\text{Größe der Hash-Tabelle}} = \frac{|S|}{m} = \frac{n}{m}$$

**Annahme:** Tabellengröße  $m$  ist fest

## Implementierung in Java

```
class TableEntry {
    private Object key,value;
}

abstract
class HashTable {
    private TableEntry[] tableEntry;
    private int capacity;

    /* Konstruktor */
    HashTable (int capacity) {
        this.capacity = capacity;
        tableEntry = new TableEntry [capacity];
        for (int i = 0; i <= capacity-1; i++)
            tableEntry[i] = null;
    }

    /* die Hashfunktion */
    protected abstract int h (Object key);

    /* fuege Element mit Schlüssel key und Wert
       value ein (falls nicht vorhanden) */
    public abstract void insert (Object key, Object
        value);

    /* entferne Element mit Schlüssel key (falls
       vorhanden) */
    public abstract void delete (Object key);

    /* suche Element mit Schlüssel key */
    public abstract Object search (Object key);
} // class hashTable
```

## Wahl der Hash-Funktion

- leichte und schnelle Berechenbarkeit
- gleichmäßige Verteilung der Daten  
(Beispiel: Compiler)

## Divisions-Rest-Methode

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

Wahl von  $m$ ?

Beispiele:

- a)  $m$  gerade  
 $\Rightarrow h(k)$  gerade  $\Leftrightarrow k$  gerade

Problematisch, wenn letztes Bit Sachverhalt ausdrückt  
(z.B. 0 = weiblich, 1 = männlich)

- b)  $m = 2^p$   
 $\Rightarrow h(k)$  liefert  $p$  niedrigsten Dualziffern von  $k$

**Regel:** Wähle  $m$  prim wobei  $m$  keine Zahl  $r^i \pm j$  teilt,  
wobei  $i$  und  $j$  kleine, nichtnegative Zahlen und  $r$   
Radix der Darstellung sind.

# Multiplikative Methode

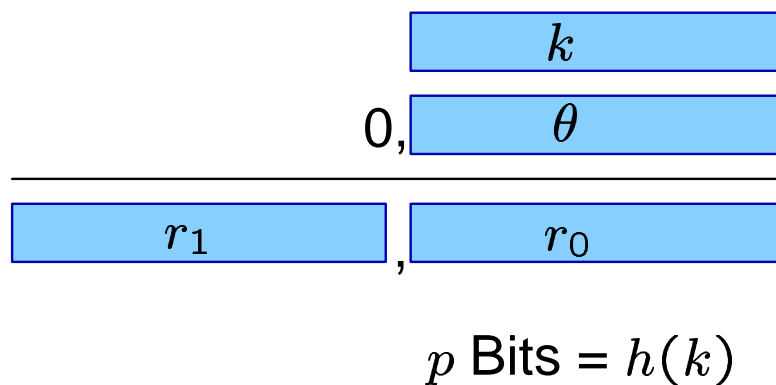
Wähle Konstante  $\theta$ ,  $0 < \theta < 1$

1. Berechne  $k\theta \bmod 1 = k\theta - \lfloor k\theta \rfloor$

2.  $h(k) = \lfloor m(k\theta \bmod 1) \rfloor$

Wahl von  $m$  unkritisch, wähle  $m = 2^p$ :

Berechnung von  $h(k)$ :



Beispiel:

$$\theta = \frac{\sqrt{5} - 1}{2} \approx 0.6180339$$

$$k = 123456$$

$$m = 10000$$

$$\begin{aligned} h(k) &= \lfloor 10000(123456 \cdot 0.61803 \dots \bmod 1) \rfloor \\ &= \lfloor 10000(76300,0041151 \dots \bmod 1) \rfloor \\ &= \lfloor 41.151 \dots \rfloor = 41 \end{aligned}$$

Von allen Zahlen  $0 \leq \theta \leq 1$  führt  $\frac{\sqrt{5}-1}{2}$  zur gleichmäßigsten Verteilung.

### 3. Universelles Hashing

**Problem:**  $h$  fest gewählt  $\Rightarrow$  es gibt  $S \subseteq \mathcal{U}$  mit vielen Kollisionen

**Idee des universellen Hashing:**

Wähle Hashfunktion  $h$  **zufällig**

$\mathcal{H}$  endliche Menge von Hashfunktionen

$$h \in \mathcal{H} : \mathcal{U} \longrightarrow \{0, \dots, m - 1\}$$

**Definition:**  $\mathcal{H}$  heißt **universell**, wenn für beliebige  $x, y \in \mathcal{U}$  gilt:

$$\frac{|\{h \in \mathcal{H} \mid h(x) = h(y)\}|}{|\mathcal{H}|} \leq \frac{1}{m}$$

**Folgerung:**  $x, y \in \mathcal{U}$  beliebig,  $\mathcal{H}$  universell,  $h \in \mathcal{H}$  zufällig

$$Pr_{\mathcal{H}}(h(x) = h(y)) \leq \frac{1}{m}$$

# Universelles Hashing

## Definition

$$\delta(x, y, h) = \begin{cases} 1 & \text{falls } h(x) = h(y) \text{ und } x \neq y \\ 0 & \text{sonst} \end{cases}$$

Erweiterung von  $\delta$  auf Mengen  $S \subseteq \mathcal{U}$  und  $\mathcal{G} \subseteq \mathcal{H}$ :

$$\delta(x, S, h) = \sum_{s \in S} \delta(x, s, h)$$

$$\delta(x, y, \mathcal{G}) = \sum_{h \in \mathcal{G}} \delta(x, y, h)$$

**Folgerung:**  $\mathcal{H}$  ist universell, wenn für alle  $x, y \in \mathcal{U}$

$$\delta(x, y, \mathcal{H}) \leq \frac{|\mathcal{H}|}{m}$$

## Universelles Hashing – Beispiel

Hashtabelle  $T$  der Größe 3,  $|\mathcal{U}| = 5$

Betrachte die 20 Funktionen (Menge  $\mathcal{H}$ ):

$$\begin{array}{cccc} x + 0 & 2x + 0 & 3x + 0 & 4x + 0 \\ x + 1 & 2x + 1 & 3x + 1 & 4x + 1 \\ x + 2 & 2x + 2 & 3x + 2 & 4x + 2 \\ x + 3 & 2x + 3 & 3x + 3 & 4x + 3 \\ x + 4 & 2x + 4 & 3x + 4 & 4x + 4 \end{array}$$

jeweils  $(\text{mod } 5)$   $(\text{mod } 3)$

und die Schlüssel 1 und 4

Es gilt:

$$\begin{array}{l} (1 \cdot 1 + 0) \text{ mod } 5 \text{ mod } 3 = 1 = (1 \cdot 4 + 0) \text{ mod } 5 \text{ mod } 3 \\ (1 \cdot 1 + 4) \text{ mod } 5 \text{ mod } 3 = 0 = (1 \cdot 4 + 4) \text{ mod } 5 \text{ mod } 3 \\ (4 \cdot 1 + 0) \text{ mod } 5 \text{ mod } 3 = 1 = (4 \cdot 4 + 0) \text{ mod } 5 \text{ mod } 3 \\ (4 \cdot 1 + 4) \text{ mod } 5 \text{ mod } 3 = 0 = (4 \cdot 4 + 4) \text{ mod } 5 \text{ mod } 3 \end{array}$$

# Eine universelle Klasse von Hashfunktionen

## Annahmen:

- $|\mathcal{U}| = p$  mit Primzahl  $p$  und  $|\mathcal{U}| = \{0, \dots, p - 1\}$
- Seien  $a \in \{1, \dots, p - 1\}$  und  $b \in \{0, \dots, p - 1\}$  und  $h_{a,b} : \mathcal{U} \rightarrow \{0, \dots, m - 1\}$  wie folgt definiert:

$$h_{a,b} = ((ax + b) \bmod p) \bmod m$$

## Folgerung:

Die Menge

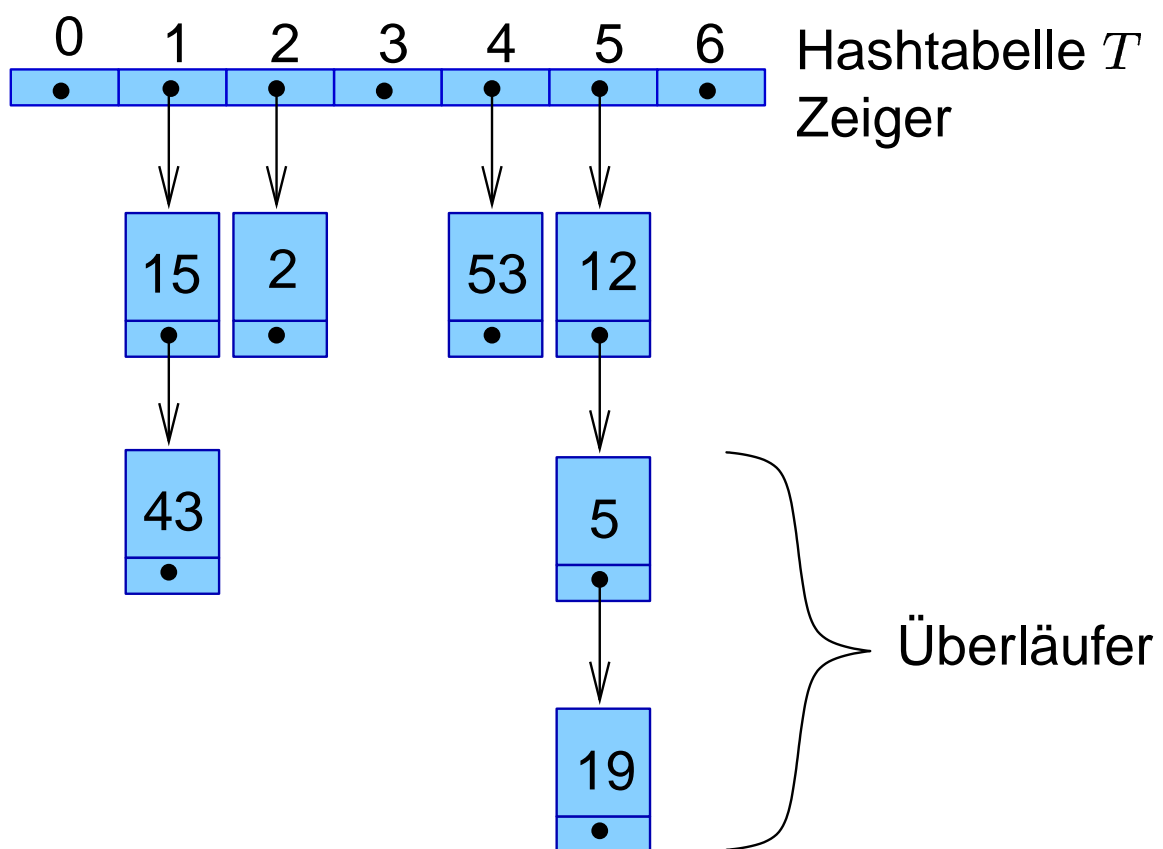
$$\mathcal{H} = \{h_{a,b} \mid 1 \leq a \leq p, 0 \leq b \leq p\}$$

ist eine **universelle Klasse von Hashfunktionen**.

## Hashing mit Verkettung der Überläufer

Schlüssel werden in **Überlauflisten** gespeichert

$$h(k) = k \bmod 7$$



Diese Art der Verkettung wird auch als **direkte Verkettung** bezeichnet.

# Verkettung der Überläufer

## Suchen nach Schlüssel $k$

- Berechne  $h(k)$  und Überlaufliste  $T[h(k)]$
- Suche nach  $k$  in der Überlaufliste

## Einfügen eines Schlüssels $k$

- Suchen nach  $k$  (erfolglos)
- Einfügen in die Überlaufliste

## Entfernen eines Schlüssels $k$

- Suchen nach  $k$  (erfolgreich)
- Entfernen aus Überlaufliste

⇒ Reine Listenoperationen

## Implementierung in Java

```
class ChainedTableEntry extends TableEntry {
    /* Konstruktor */
    ChainedTableEntry(Object key, Object value) {
        super(key, value);
        this.next = null;
    }
    private ChainedTableEntry next;
}

class ChainedHashTable extends HashTable {
    /* die Hashfunktion */
    public int h(Object key) {
        return key.hashCode() % capacity ;
    }

    /* suche key in der Hashtabelle */
    public Object search (Object key) {
        ChainedTableEntry p;
        p = (ChainedTableEntry) tableEntry[h(key)];

        /* Gehe die Liste durch bis Ende erreicht oder
           key gefunden */
        while (p != null && !p.key.equals(key)) {
            p = p.next;
        }

        /* Gib Ergebnis zurück */
        if (p != null)
            return p.value;
        else return null;
    }
}
```

## Implementierung in Java

```
/* fuege ein Element mit Schlüssle key und Wert
   value ein (falls nicht vorhanden) */
public void insert (Object key, Object value) {

    ChainedTableEntry entry = new
                                ChainedTableEntry(key,
                                value);

    /* Hole den Tabelleneintrag für key */
    int k = h (key);
    ChainedTableEntry p;
    p = (ChainedTableEntry) tableEntry [k];

    if (p == null){
        tableEntry[k] = entry;
        return ;
    }
    /* Suche nach key */
    while (!p.key.equals(key) && p.next != null) {
        p = p.next;
    }

    /* Fuege das Element ein (falls nicht vorhanden)
       */
    if (!p.key.equals(key))
        p.next = entry;
}
```

## Implementierung in Java

```
/* entferne das Element mit Schluessel key (falls
   vorhanden) */
public void delete (Object key) {
    int k = h (key);
    ChainedTableEntry p;
    p = (ChainedTableEntry) TableEntry [k];
    TableEntry[k] = recDelete(p, key);
}

/* entferne das Element mit Schluessel key
   rekursiv (falls vorhanden) */
public ChainedTableEntry
recDelete (ChainedTableEntry p, Object key) {
    /* recDelete gibt einen Zeiger auf den Beginn
       der Liste, auf die p zeigt, zurueck, in der
       key entfernt wurde */
    if (p == null)
        return null;
    if (p.key.equals(key))
        return p.getNext();
    /* ansonsten: */
    p.next = recDelete(p.next, key);
    return p;
}

public void printTable () {...}

} // class ChainedHashTable
```

# Test-Programm

```
public class ChainedHashingTest {
    public static void main(String args[]){
        Integer[] t= new Integer[args.length];

        for (int i = 0; i < args.length; i++)
            t[i] = Integer.valueOf(args[i]);

        ChainedHashTable h = new ChainedHashTable(7);
        for (int i = 0; i <= t.length - 1; i++)
            h.insert(t[i], null);

        h.printTable ();
        h.delete(t[0]); h.delete(t[1]);
        h.delete(t[6]); h.printTable();
    }
}
```

## Aufruf:

```
java ChainedHashingTest 12 53 5 15 2 19 43
```

## Ausgabe:

```
0: -|          0: -|
1: 15 -> 43 -|  1: 15 -|
2: 2 -|         2: 2 -|
3: -|          3: -|
4: 53 -|       4: -|
5: 12 -> 5 -> 19 -| 5: 5 -> 19 -|
6: -|          6: -|
```

# Analyse der direkten Verkettung

## Uniform-Hashing Annahme:

- alle Hashadressen werden mit gleicher Wahrscheinlichkeit gewählt, d.h.:  
 $Pr(h(k_i) = j) = 1/m$
- unabhängig von Operation zu Operation

Mittlere Kettenlänge bei  $n$  Einträgen:

$$n/m = \alpha$$

## Definition

$C'_n$  = Erwartungswert für die Anzahl betrachteter Einträge bei **erfolgloser** Suche

$C_n$  = Erwartungswert für die Anzahl betrachteter Einträge bei **erfolgreicher** Suche

## Analyse

$$C'_n = \alpha$$
$$C_n \approx 1 + \frac{\alpha}{2}$$

# Verkettung der Überläufer

## Vorteile:

- +  $C_n$  und  $C'_n$  niedrig
- +  $\alpha > 1$  möglich
- + echte Entfernungen
- + für Sekundärspeicher geeignet

## Effizienz der Suche

$\alpha$	$C_n$ (erfolgreich)	$C'_n$ (erfolglos)
0.50	1.250	0.50
0.90	1.450	0.90
0.95	1.475	0.95
1.00	1.500	1.00
2.00	2.000	2.00
3.00	2.500	3.00

## Nachteile:

- Zusätzlicher Speicherplatz für Zeiger
- Überläufer außerhalb der Hashtabelle