Praktische Informatik 2
Hashing

Thomas Röfer

Cyber-Physical Systems
Deutsches Forschungszentrum für
Künstliche Intelligenz

Multisensorische Interaktive Systeme
Fachbereich 3, Universität Bremen

Bild: ©Michael Ihle

Universität Bremen

Motivation

- Der Zugriff auf Elemente in sortierten Arrays oder balancierten Bäumen benötigt im Mittel **O(log n)**
- Der Zugriff auf Elemente eines Arrays (bei bekanntem Index) benötigt aber nur **O(1)**
- Schön wäre es, wenn man aus dem Suchwert **direkt** den **Index** eines Datensatzes in einem Array **berechnen** könnte → **Hashing**

Hash-Verfahren (Streuspeicherverfahren)

- Hash-Funktion
  - Bildet Objekte auf ganze Zahlen (**Hashcodes**) ab
  - Hashcodes werden als numerische Schlüssel zur Identifikation der Objekte genutzt
- Hash-Tabelle
  - Ein Array, in dem die Objekte eingetragen werden
  - Dazu wird der Hashcode als Index genutzt, z.B.
    hashTable[object.hashCode()] = object;

Beispiel: Personaldatenbank

- Als Schlüssel wird Anfangsbuchstabe des Nachnamens genutzt
- Jedem dieser Buchstaben kann eine Zahl (z.B. Position im Alphabet) zugeordnet werden
- Diese Zahl ist der Hashcode des Objekts „Person“
- Problem: Gleicher Hashcode unterschiedlicher Objekte
  - Besserer Hashcode (hilft nur begrenzt)
  - Behandlung von Überläufen (**Kollisionen**)

Anton Wagner	W	23
Doris Bach	B	2
Doris May	M	13
Friedrich Dörig	D	4

1	
2	Doris Bach
3	
4	Friedrich Dörig
5	
6	
7	
8	
9	
10	
11	
12	
13	Doris May
:	:
23	Anton Wagner
:	:

Universität Bremen

Hash-Funktionen

h(„Doris Bach") = 2

- **h** : Menge der Objekte $\rightarrow \mathbb{N}$
- Kodieren komplizierter Objekte durch „Zerhacken" (Hashing) in eine kleine Zahl, die als Schlüssel dienen kann
- Sollte vom gesamten Datensatz abhängen, so dass sie für unterschiedliche Datensätze möglichst auch unterschiedliche Hash-Codes liefert
- Sollte für alle tatsächlich vorkommenden Objekte möglichst gleich verteilt sein, d.h. die Hash-Codes sollten etwa gleich oft vorkommen
- Berechnung des Hash-Wertes sollte nicht zu lange dauern
- Modulare Hash-Funktion: **h(k) = k mod m**

Universität Bremen

$(a + b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m$
$(a \cdot b) \bmod m = ((a \bmod m) \cdot (b \bmod m)) \bmod m$

Zeichenkette zu Hashcode

- Betrachtung der Zeichen als Stellen einer Zahl, z.B. als Ziffern zur Basis 256
- Beispiel: **„INFO“** → **(73, 78, 70, 79)**
  - **$h(„INFO“) = (73 \cdot 256^3 + 78 \cdot 256^2 + 70 \cdot 256^1 + 79 \cdot 256^0) \bmod m$**
  - **m** ist die Anzahl der Einträge der Hash-Tabelle
- Problem: Zwischenergebnisse werden bei längeren Strings zu groß für 32 Bit
  - **$h(„INFORMATIK“) = (73 \cdot 256^9 + 78 \cdot 256^8 + \ldots + 75 \cdot 256^0) \bmod m$**
- Lösung: Man kann schreiben (Horner-Schema)

$$\begin{aligned} h(„INFORMATIK“) &= (\ldots((73 \cdot 256) + 78) \cdot 256) + \ldots + 75) \bmod m \\ &= (\ldots((((73 \cdot 256) \bmod m) + 78) \bmod m) \cdot 256) \bmod m + \ldots + 75) \bmod m \end{aligned}$$

Universität Bremen

Hash-Funktion für Zeichenketten: Beispiel

```
int hash(final String s, final int m)
{
    final int a = 256; // oder 65536 für Unicode
    int h = 0;
    for (int i = 0; i < s.length(); ++i) {
        h = (h * a + s.charAt(i)) % m;
    }
    return h;
}
```

Universität Bremen

Hash-Funktionen in Java

- Klasse **Object** deklariert bereits Methode **int hashCode()**
- Sie generiert Hash-Codes im Wertebereich von **int**
- Ihr Wert muss noch **positiv** gemacht und **modulo Tabellenlänge** verrechnet werden (**Math.floorMod**)
- Implementierung in **Object** erzeugt Hash-Code aus Adresse des Objekts
- **hashCode()** muss so überschrieben werden, dass für gleiche Objekte (im Sinne von **equals**) derselbe Hash-Code geliefert wird

Eigentlich ist die Obergrenze hier sinnlos (wegen Überlaufs)

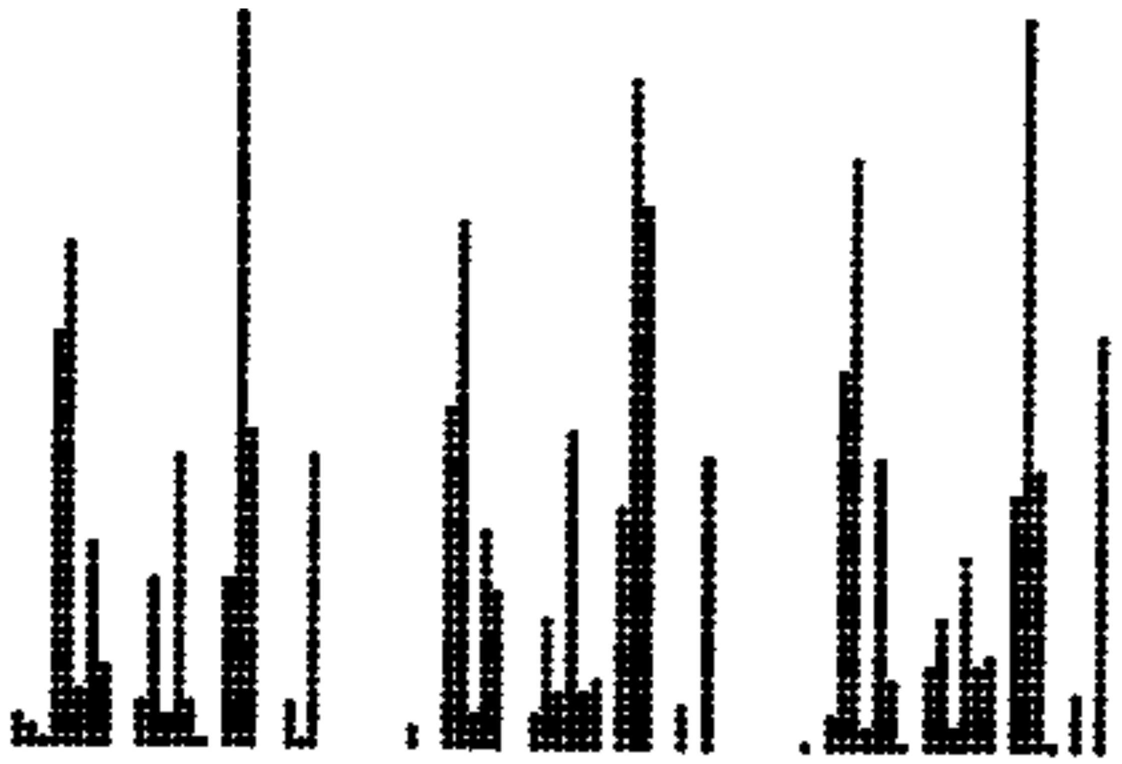
```
class Person
{
    private String name;
    // …
    public int hashCode()
    {
        return hash(name,
                Integer.MAX_VALUE);
    }

    public boolean equals(
            final Person p)
    {
        return name.equals(p.name);
    }
}
```

Hash-Funktionen: Wahl von **m** und **a**

- Ziel: Zufällig verteilte Schlüssel auch bei nicht zufällig verteilten Daten
- Beispiel: Moby Dick von Herman Melville (Englisch, 1000 Wörter)
- Ungünstig: **a = m** → nur letztes Zeichen bestimmt Hashcode
- Ungünstig: **a = m · n, n** $\in \mathbb{N}$
- Günstig: **a** ist Primzahl und/oder **m** ist Primzahl
- Günstig: **a** und **m** sind teilerfremd

m = 96, a = 128



m = 97, a = 128

m = 96, a = 127

$$h(X) = (\dots ((x_0 \cdot a) \bmod m) + \dots + x_n) \bmod m$$

Universität Bremen

Universelle Hash-Funktion

- Theoretisch ideal, wenn Wahrscheinlichkeit einer Kollision zwischen zwei Schlüsseln **1/m** ist

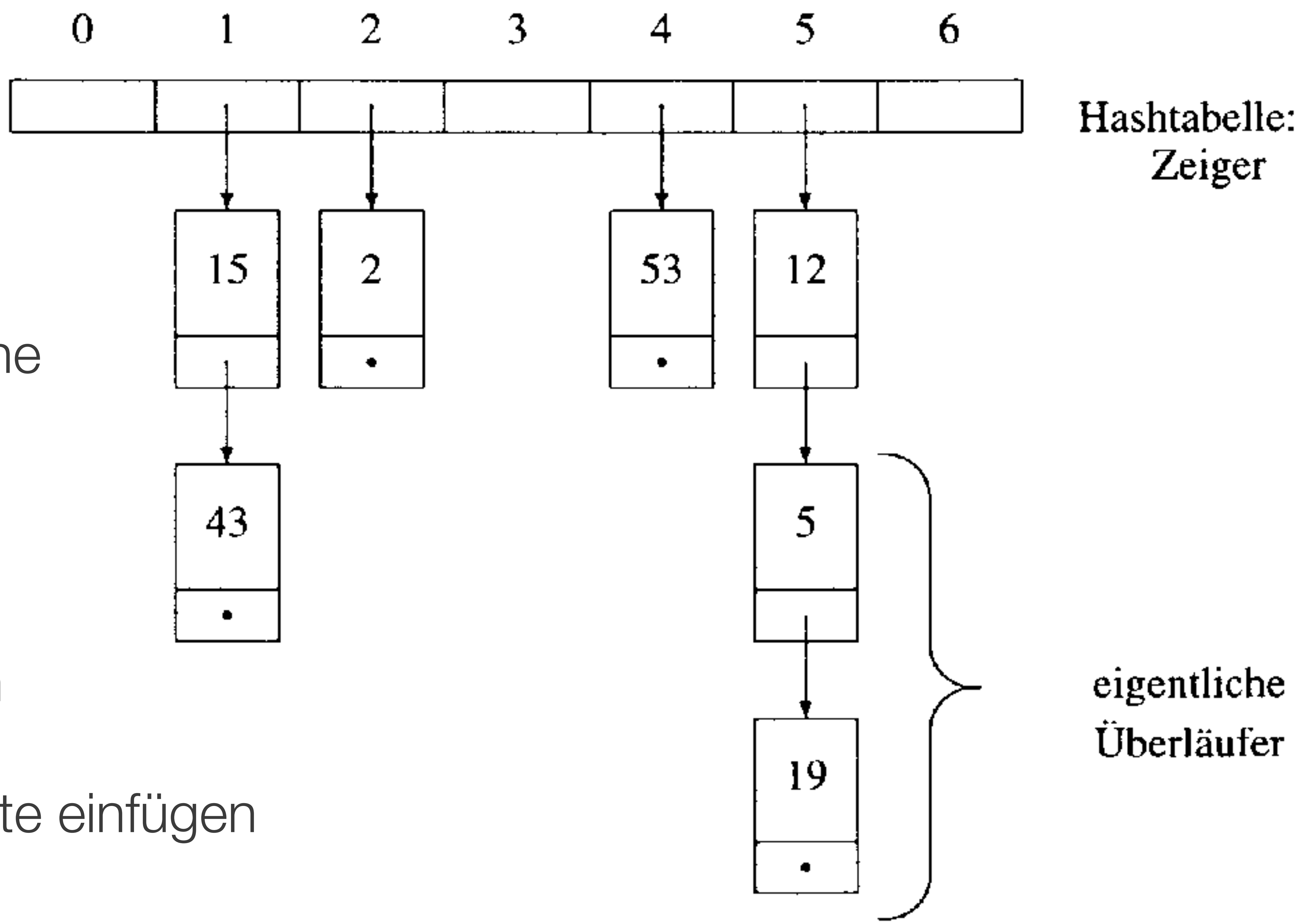
- Ansatz für Strings: Wenn **a** von **m** unabhängig sein soll, wähle pseudo-zufälliges **a**, d.h. für jede Stelle der Zeichenkette wird ein anderes **a** verwendet

```
int hashU(final String s, final int m)
{
    int a = 31415;
    final int b = 27183;
    int h = 0;
    for (int i = 0; i < s.length(); ++i) {
        h = (h * a + s.charAt(i)) % m;
        a = a * b % (m – 1);
    }
    return h;
}
```

- $a_{n+1} = a_n \cdot b \bmod (m - 1)$ erzeugt pseudo-zufällige Zahlen im Bereich **[0 … m-2]**

- Funktioniert nur, wenn **m-1** kein Teiler von **a** oder **b** ist

Verkettung der Überläufer



- Suchen: Beginne bei **HT[h(k)]** und durchsuche (z.B. einfach verkettete) Liste
  - Falls gefunden → zurückliefern
  - Falls Listenende erreicht → nicht gefunden
- Einfügen: Bei **HT[h(k)]** an den Anfang der Liste einfügen
- Löschen: Suchen und wenn gefunden, aus Liste entfernen
- Aufwand: Bei Gleichverteilung der Werte auf die Hash-Tabelle: **O(n / m)**
  - Hashing lohnt sich also, wenn **m** im Verhältnis zu **n** nicht zu klein ist

Offenes Hashing (Hashing mit offener Adressierung)

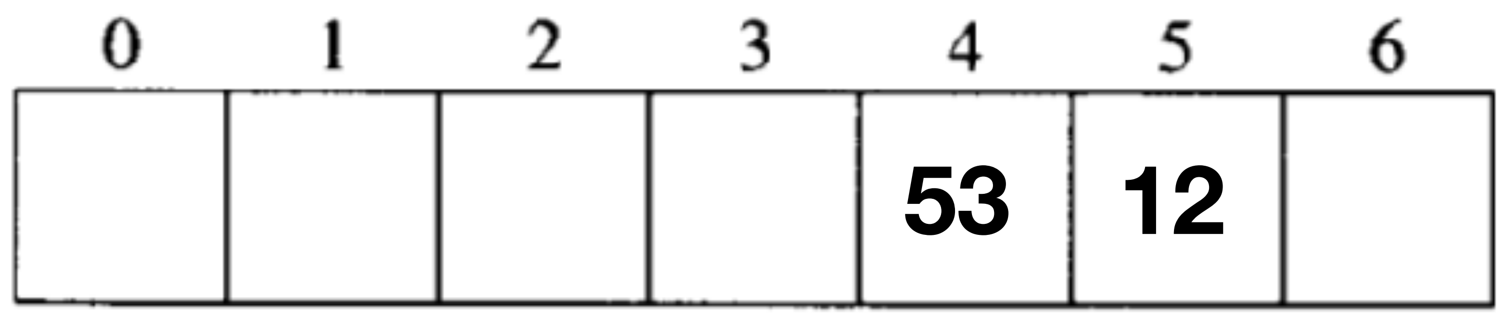
- Motivation
  - Verkettung der Überläufer benötigt zusätzlichen Speicher
  - Belegen (und Freigeben) des zusätzlichen Speichers kostet Zeit
- Alle Werte werden in der Hash-Tabelle selbst gespeichert
- Bei einer **Kollision** wird der Wert in einem Ausweichplatz gespeichert
- Die Suche nach Ausweichplätzen heißt **Sondieren**

Sondieren

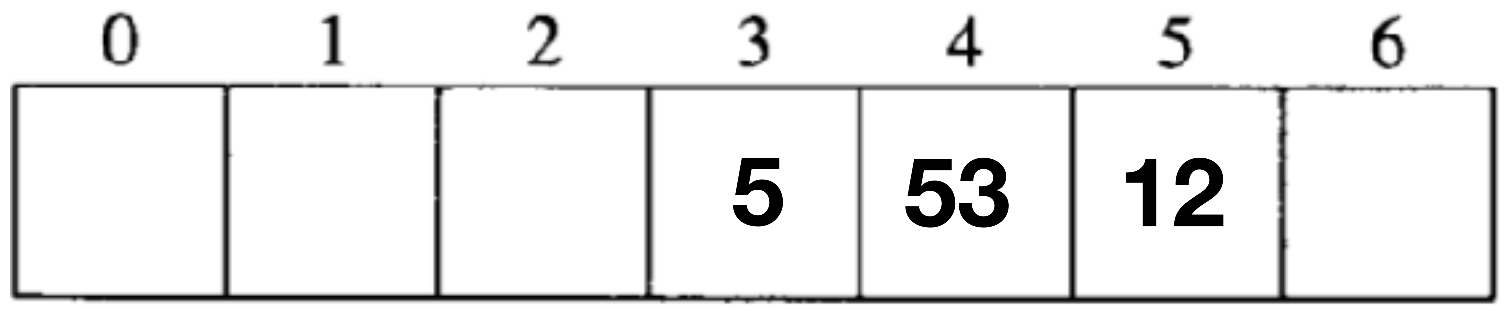
- Sondierungsfunktion **$s(j, k)$**
  - Liefert Versatz zum Hash-Wert, d.h. es werden Indizes **$(h(k) - s(j, k)) \bmod m$** durchsucht
  - **j**: Anzahl der Fehlversuche im Bereich **$0 \dots m-1$**
- Lineares Sondieren
  - Hash-Tabelle wird linear nach Werten durchsucht:
    $h(k), h(k) - 1, h(k) - 2, \dots, 0, m - 1, \dots h(k) + 1$
  - Sondierungsfunktion: **$s(j, k) = j$**

Lineares Sondieren: Beispiel

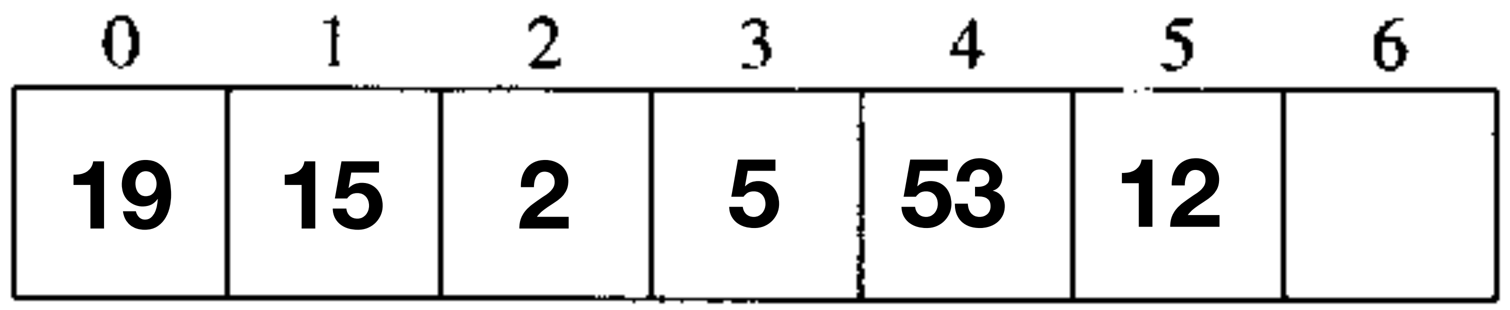
- **$m = 7$, $h(k) = k \bmod m$, $s(j, k) = j$**
- Einfügen von **12**, **53**



- Einfügen von **5**
  - Sondierungsfolge **5-4-3**



- Einfügen von **15**, **2**, **19**
  - Sondierungsfolge **5-4-3-2-1-0**



Primäre Häufung

- Belegungsfaktor α **=** Anzahl der belegten Elemente **/ m**

- Nachteile
  - Häufungspunkte senken die Effizienz (**Primäre Häufung**)
  - Aufwand steigt erheblich, wenn α gegen **1** geht

- Beispiel: Nach dem Einfügen von **12**, **53**, **5** würden weitere Schlüssel folgendermaßen abgelegt:
  - Werte mit **h(k) = 1** landen in **HT[1]**
  - Werte mit **h(k) = 2…5** landen in **HT[2]**

0	1	2	3	4	5	6
			5	53	12	

Quadratisches Sondieren

- Hash-Tabelle wird quadratisch nach Werten durchsucht
  - **$s(j, k) = \lceil j / 2 \rceil^2 \cdot (-1)^j$**
  - **$h(k), h(k) + 1, h(k) - 1, h(k) + 4, h(k) - 4, \ldots$**
- Beispiel: Einfügen von **12**, **53**, **5**, **15**, **2**
  - Sondierungsfolge für **5**: **h(5)**, **h(5)+1**
- Einfügen von **19**
  - **h(19) = 5**, **5 + 1**, **5 - 1**, **(5 + 4) mod 7 = 2**, **5 – 4 = 1**, **(5 + 9) mod 7 = 0**

0	1	2	3	4	5	6
	15	2		53	12	5

0	1	2	3	4	5	6
19	15	2		53	12	5

Sekundäre Häufung und Sondierungsaufwand

- **Sekundäre Häufung**: Synonyme behindern sich gegenseitig, z.B. **5** und **19**
- Mittlerer Aufwand lineares Sondieren
  - Erfolglose Suche: $\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$ z.B. für α = **0,75**: **8,5**
  - Erfolgreiche Suche: $\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$ z.B. für α = **0,75**: **2,5**
- Mittlerer Aufwand quadratisches Sondieren
  - Erfolglose Suche: $\frac{1}{1-\alpha}-\alpha+\ln\left(\frac{1}{1-\alpha}\right)$ z.B. für α = **0,75**: **4,6**
  - Erfolgreiche Suche: $1-\frac{\alpha}{2}+\ln\left(\frac{1}{1-\alpha}\right)$ z.B. für α = **0,75**: **2,0**

Doppeltes Hashing

- Auch mit quadratischem Sondieren stören sich Werte mit demselben Hashcode **$h(k)$**
- Für das Sondieren wird zweite Hash-Funktion **$h'(k)$** verwendet: **$s(j, k) = j \cdot h'(k)$**
  - **$h(k), h(k) - h'(k), h(k) - 2 \cdot h'(k), \ldots, h(k) - (m - 1) \cdot h'(k)$**
- Anforderungen an **$h'(k)$**
  - **$h'(k) \neq 0$**
  - Darf kein Teiler von **m** sein (erfüllt, wenn **m** eine Primzahl ist)
  - Sollte unabhängig von **$h(k)$** sein: **$p[h(k) = h(k') \wedge h'(k) = h'(k')] = p[h(k) = h(k')] \cdot p[h'(k) = h'(k')]$**
- Gute Wahl für **$h'(k)$** falls **m** eine Primzahl ist: **$h'(k) = 1 + k \bmod (m - 2)$**

Doppeltes Hashing: Beispiel

- **$m = 7$, $h(k) = k \bmod m$, $h'(k) = 1 + k \bmod (m - 2)$, $s(j, k) = j \cdot h'(k)$**
- Einfügen von **12**, **53**

0	1	2	3	4	5	6
				53	12	

- Einfügen von **5**, **15**, **2**
  - Sondierungsreihenfolge für **5** ist
    $h(5) = 5 \bmod 7 = 5$, $5 - (1 + 5 \bmod 5) = 4$, $5 - 2 = 3$

0	1	2	3	4	5	6
	15	2	5	53	12	

- Einfügen von **19**
  - Sondierungsreihenfolge ist
    $h(19) = 19 \bmod 7 = 5$, $5 - (1 + 19 \bmod 5) = 0$

0	1	2	3	4	5	6
19	15	2	5	53	12	

Verbesserung der erfolgreichen Suche

- Einfügereihenfolge **12**, **53**, **5**, **15**, **13**, **19**

0	1	2	3	4	5	6
19	15		5	53	12	13

  - **(Suchzeit(12) + Suchzeit(53) + Suchzeit(5) + Suchzeit(15) + Suchzeit(13) + Suchzeit(19)) / 6 = 9 / 6 = 1,5**

- Einfügereihenfolge **53**, **5**, **15**, **13**, **19**, **12**

0	1	2	3	4	5	6
19	15	12		53	5	13

  - Durchschnittliche Suchzeit: **8 / 6 = 1,33…**

- Durchschnittliche Suchzeit hängt von der Reihenfolge des Einfügens ab

Brents Algorithmus

- Einfügen von **12**, **53**

0	1	2	3	4	5	6
				53	12	

- Einfügen von **5** (**5** und **12** austauschen, **12** einfügen)

0	1	2	3	4	5	6
		12		53	5	

- Mittlerer Aufwand
  - Erfolgloses Suchen: $$\frac{1}{1-\alpha}$$
  - Erfolgreiches Suchen: $$1 + \frac{\alpha}{2} + \frac{\alpha^3}{4} + \frac{\alpha^4}{15} + \ldots < 2{,}5$$

```
void brentInsert(final T k)
{
    int i = h(k);
    while (ht[i] != null) {
        final int b = (i – h_(k)) % ht.length;
        final int bb = (i – h_(ht[i])) % ht.length;
        if (ht[b] != null && ht[bb] == null) {
            final T kk = k; k = ht[i]; ht[i] = kk;
            i = bb;
        }
        else {
            i = b;
        }
    }
    ht[i] = k;
}
```

Zusammenfassung der Konzepte

- **Hash-Funktion**
- **Hashing mit Verkettung**
- **Offenes Hashing**
  - **Sondieren** (**linear**, **quadratisch**, **doppeltes Hashing**)