



# Hashing

Thomas Röfer

Hash-Funktionen

Hashing mit Verkettung

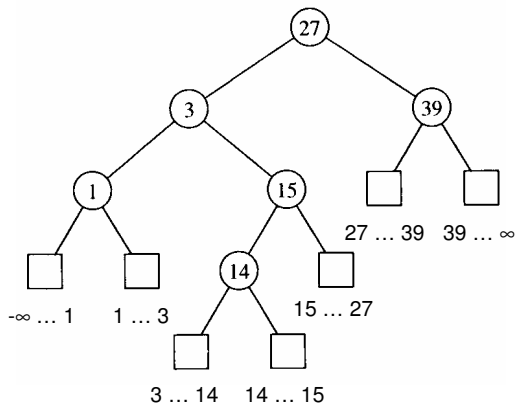
Offenes Hashing

Doppeltes Hashing

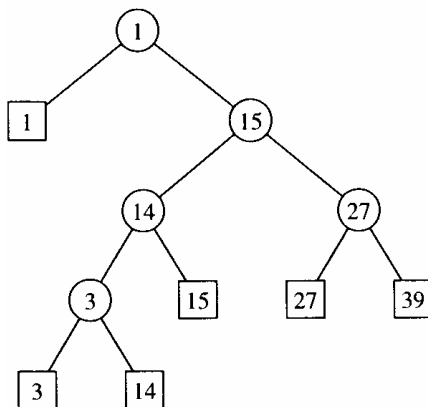
Dynamische Hash-Tabellen

# Rückblick „Bäume 2“

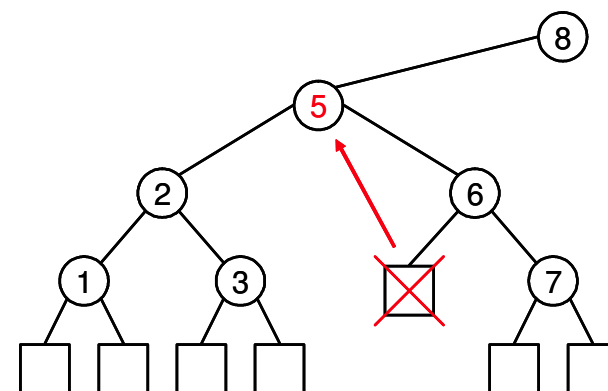
Suchbäume



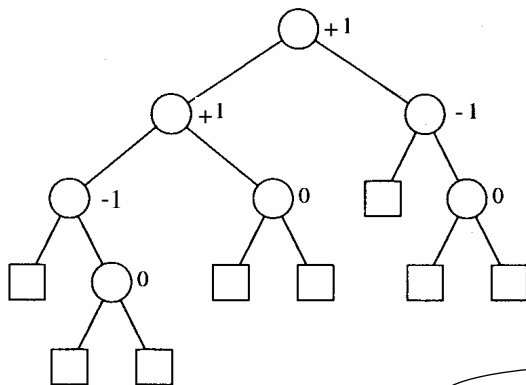
Blattsuchbäume



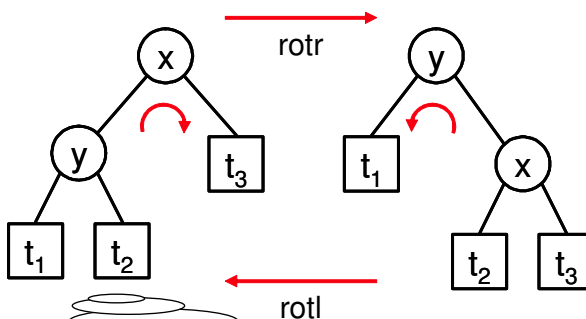
Löschen aus Suchbaum



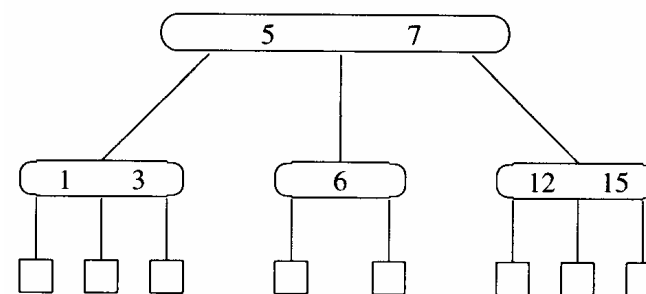
AVL-Bäume



Rotieren von Knoten



B-Bäume



# Organisatorisches

## ▶ **Übungsblatt 5**

- ▶ Ausgabe 17.05
- ▶ Abgabe 01.06 (Dienstag), 10:00-11:30 in MZH 3060
- ▶ Tutorien 17.05 (Mo), 18.05 (Di), 27.05 (Do)

## ▶ **Übungsblatt 6**

- ▶ Ausgabe 24.05
- ▶ Abgabe 07.06
- ▶ Tutorien 24.05 (Mo), 25.05 (Di), 03.06 (Do)

## ▶ **Übungsblatt 7**

- ▶ Ausgabe 07.06
- ▶ Abgabe 14.06

## ▶ **Sonstiges**

- ▶ Eclipse-Vorstellung 01.06 (Di), 9:00 s.t.-10:00 in MZH 7210



# Überblick

## ▶ 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

## ▶ Streuspeicherverfahren (Hash-Verfahren)

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



# Beispiel – Mitarbeiterdatenbank

## ▶ Hashcode

- ▶ Als Schlüssel wird der Anfangsbuchstabe des Nachnamens genutzt
- ▶ Jedem dieser Buchstaben kann eine Zahl (z.B. Position im Alphabet) zugeordnet werden
- ▶ Diese Zahl ist der Hashcode des Objekts „Mitarbeiter“

## ▶ Beispiel

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

## ▶ Problem: Gleicher Hashcode unterschiedlicher Objekte

- ▶ Besserer Hashcode (hilft nur begrenzt)
- ▶ Behandlung von Überläufen (*Kollisionen*)

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

:

23	Anton Wagner
----	--------------

:



# Hash-Funktionen

## ▶ Definition

- ▶  $h$  : Menge der Objekte  $\rightarrow \mathbb{N}$
- ▶ Ziel ist das kodieren komplizierter Objekte durch „Zerhacken“ (Hashing) in eine kleine Zahl, die als Schlüssel dienen kann

## ▶ Beispiel

- ▶  $h(\text{„Doris Bach“}) = 2$
- ▶ „Doris Bach“ wird also an Index 2 in der Hashtabelle gespeichert
  - ▶  $HT[2] = \text{„Doris Bach“}$

## ▶ Ziele

- ▶ Die Hash-Funktion sollte vom gesamten Datensatz abhängen, so dass sie für unterschiedliche Datensätze möglichst auch unterschiedliche Hash-Codes liefert
- ▶ Die Hash-Funktion 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

- ▶  $h(k) = k \bmod m$



# Von der Zeichenkette zum Hashcode

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



# Hash-Funktion in Java

## ▶ Für Strings

- ▶ Implementierung in Schleife
- ▶ Eigentlich  $a = 65536$  für Unicode

## ▶ hashCode()

- ▶ Die Klasse *Object* deklariert bereits eine Objektmethode *int hashCode()*
- ▶ Sie sollte genutzt werden, um Hashcodes im Wertebereich von *int* zu erzeugen
- ▶ Bei der Benutzung muss ihr Wert noch *modulo* Tabellenlänge verrechnet werden
- ▶ Die Implementierung in *Object* nutzt die Adresse des Objekts als Hashcode
- ▶ Daher muss *hashCode()* überschrieben werden, weil sonst gleiche Objekte unterschiedliche Hashcodes erhalten

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

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

# Hash-Funktionen

## ▶ Ziel

- ▶ Zufällig verteilte Schlüssel auch bei nicht zufällig verteilten Daten

## ▶ Beispiel: Moby Dick von Herman Melville

- ▶ Die ersten 1000 eindeutigen Wörter (in Englisch)

## ▶ Wahl von $m$ und $a$

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

- ▶ Ungünstig:  $a = m \rightarrow$  nur letztes Zeichen bestimmt den Hashcode

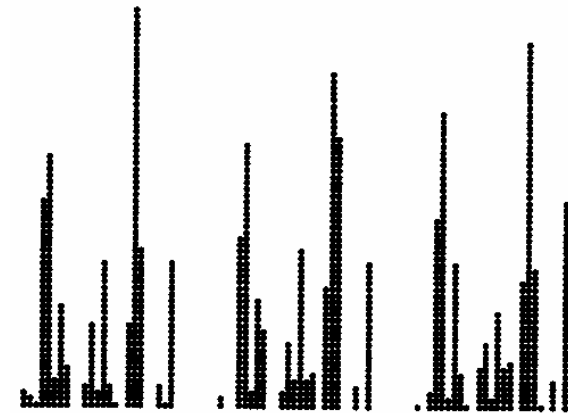
- ▶ Ebenfalls ungünstig:

$$a = m \cdot n, n \in \mathbb{N}$$

- ▶ Günstig:  $a$  ist Primzahl und/oder  $m$  ist Primzahl

- ▶ Günstig:  $a$  und  $m$  sind teilerfremd

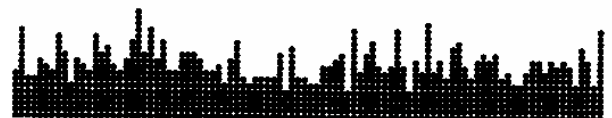
$$m = 96, a = 128$$



$$m = 97, a = 128$$



$$m = 96, a = 127$$





# Universelle Hash-Funktion

- ▶ **Universelle Hash-Funktion**
  - ▶ Theoretisch ideal, wenn die Wahrscheinlichkeit einer Kollision zwischen zwei Schlüsseln  $1/m$  beträgt
- ▶ **Ansatz für Strings**
  - ▶ Wenn  $a$  von  $m$  unabhängig sein soll, wählt man ein pseudo-zufälliges  $a$
  - ▶ D.h. für jede Stelle der Zeichenkette wird ein anderes  $a$  verwendet
- ▶ **Anmerkung**
  - ▶  $a_{n+1} = a_n \cdot b \bmod (m - 1)$  erzeugt pseudo-zufällige Zahlen im Bereich  $[0 \dots m - 2]$
  - ▶ Funktioniert nur, wenn  $m-1$  kein Teiler von  $a$  oder  $b$  ist

```
static int hashU(String s, 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;
}
```

# Verkettung der Überläufer

## Suchen

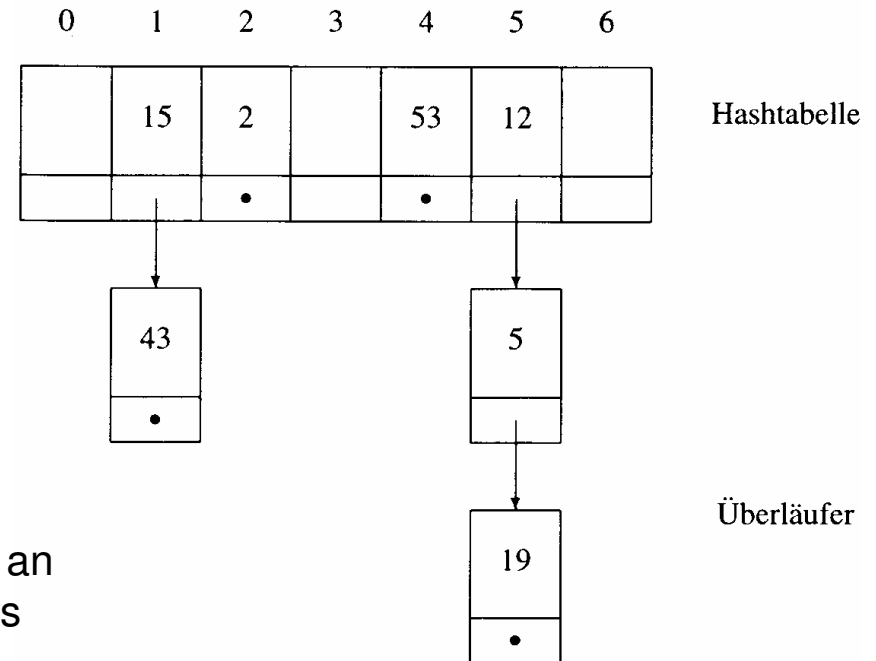
- ▶ Beginne bei  $HT(h(k))$
- ▶ Durchsuche den Eintrag und Liste aller Überläufer
- ▶ Falls gefunden  $\rightarrow$  zurückliefern
- ▶ Falls Listenende erreicht  $\rightarrow$  nicht gefunden

## Einfügen mit Überprüfung auf Doppelte

- ▶ Wert suchen
- ▶ Falls nicht gefunden, in die Hashtabelle bzw. an das Ende der Überläuferliste einfügen (da das Listenende während der Suche bereits gefunden wurde)

## Einfügen ohne Überprüfung auf Doppelte

- ▶ Direkt bei  $HT(h(k))$  bzw. an den Anfang der Überläuferliste einfügen



# Verkettung der Überläufer

## ▶ Löschen

- ▶ Wert suchen
- ▶ Falls in Tabelle gefunden, dann ersten Überläufer aus der Liste aushängen und Eintrag in Tabelle überschreiben
- ▶ Falls in Überläuferliste gefunden, dann einfach aus der Liste aushängen

## ▶ Direkte Verkettung

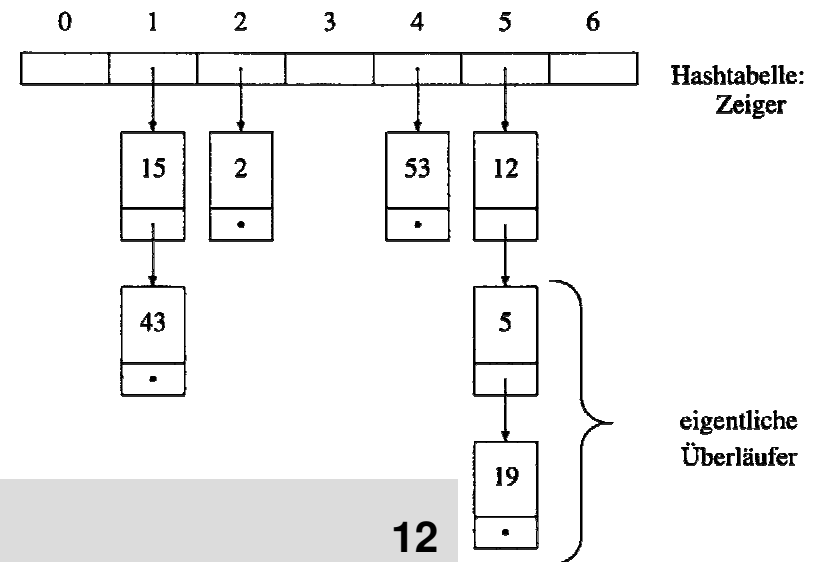
- ▶ Alle Werte stehen in der Überläuferliste
- ▶ Vorteile
  - ▶ *Einfachere Implementierung*
  - ▶ *Weniger Platzverschwendung für unbelegte Einträge in der Hashtabelle*
- ▶ Nachteile
  - ▶ *Größerer Platzverbrauch durch zusätzliche Referenz*
  - ▶ *Etwas langsamer im Zugriff*

## ▶ Aufwand

- ▶ Bei Gleichverteilung der Werte auf die Hashtabelle:  $O(n / m)$
- ▶ Hashing lohnt sich also, wenn  $m$  im Verhältnis zu  $n$  nicht zu klein ist

## ▶ Anmerkung

- ▶ Prinzipiell können Überläufer durch jede Art von Datenstruktur verwaltet werden, also z.B. auch durch Bäume





# Offenes Hashing

## ▶ Motivation

- ▶ Bei der Verkettung der Überläufer wird zusätzlicher Speicher benötigt
- ▶ Das Belegen (und Freigeben) des zusätzlichen Speichers kostet Zeit

## ▶ Ansatz

- ▶ Alle Werte werden in der Hash-Tabelle gespeichert
- ▶ Bei einer *Kollision* wird der Wert in einem Ausweichplatz gespeichert
- ▶ Die Suche nach Ausweichplätzen heißt *Sondieren*

## ▶ Lineares Sondieren

- ▶ Hashtabelle wird linear nach Werten durchsucht
  - ▶  $h(k), h(k) - 1, h(k) - 2, \dots, 0, m - 1, \dots, h(k) + 1$
- ▶ Sondierfunktion
  - ▶  $s(j, k) = j$



# Lineares Sondieren

## ▶ Beispiel

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

0	1	2	3	4	5	6
				53	12	

- ▶ Einfügen von 5
  - ▶ Sondierungsfolge 5-4-3

0	1	2	3	4	5	6
			5	53	12	

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

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

## ▶ Begriff

- ▶ *Belegungsfaktor*
  - ▶  $\alpha = \text{Anzahl der belegten Elemente} / m$

## ▶ Nachteile

- ▶ Häufungspunkte senken die Effizienz (*Primäre Häufung*)
- ▶ Aufwand steigt erheblich, wenn  $\alpha$  gegen 1 geht

## ▶ Beispiel

- ▶ Nach dem Einfügen von 12, 53, 5 würden weitere Schlüssel landen
  - ▶ Werte mit  $h(k) = 1$  landen in  $HT[1]$
  - ▶ Werte mit  $h(k) = 2 \dots 5$  landen in  $HT[2]$



# Quadratisches Sondieren

## ▶ Sondierungsfunktion

- ▶ Hashtabelle wird quadratisch nach Werten durchsucht
  - ▶  $s(j, k) = \lceil j/2 \rceil^2 \cdot (-1)^j$
  - ▶  $h(k), h(k) + 1, h(k) - 1, h(k) + 4, h(k) - 4, \dots$

## ▶ Beispiel

- ▶ Einfügen von 12, 53, 5, 15, 2
  - ▶ *Sondierungsfolge für 5:  $h(5), h(5)+1$*

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

- ▶ Einfügen von 19
  - ▶ *Folge  $h(19) = 5, 5 + 1, 5 - 1, (5 + 4) \bmod 7 = 2, 5 - 4 = 1, (5 + 9) \bmod 7 = 0$*

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

## ▶ Nachteile

- ▶ Synonyme behindern sich gegenseitig, z.B. 5 und 19 (*Sekundäre Häufung*)

## ▶ Aufwand lineares Sondieren

- ▶ Erfolglose Suche:  $\frac{1}{2} \left( 1 + \frac{1}{(1-\alpha)^2} \right)$
- ▶ Erfolgreiche Suche:  $\frac{1}{2} \left( 1 + \frac{1}{1-\alpha} \right)$

## ▶ Aufwand quadratisches Sondieren

- ▶ Erfolglose Suche:  $\frac{1}{1-\alpha} - \alpha + \ln \left( \frac{1}{1-\alpha} \right)$
- ▶ Erfolgreiche Suche:  $1 - \frac{\alpha}{2} + \ln \left( \frac{1}{1-\alpha} \right)$



# Doppeltes Hashing

## ▶ Motivation

- ▶ Auch mit quadratischem Sondieren stören sich alle Werte mit demselben Hashcode  $h(k)$ , weil die Sondierungsfunktion  $s(j, k)$  nicht direkt von  $k$  abhängt

## ▶ Ansatz

- ▶ Für das Sondieren wird eine zweite Hash-Funktion  $h'(k)$  verwendet
  - ▶  $s(j, k) = j \cdot h'(k)$
  - ▶  $h(k), h(k) - h'(k), h(k) - 2 \cdot h'(k), \dots, h(k) - (m-1) \cdot h'(k)$

## ▶ 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
  - ▶  $p[h(k) = h(k') \text{ und } h'(k) = h'(k')] = p[h(k) = h(k')] \cdot p[h'(k) = h'(k')]$

## ▶ 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

## ▶ Beobachtung

- ▶ Die durchschnittliche Suchzeit hängt von der Reihenfolge des Einfügens der Werte ab

## ▶ Beispiel

- ▶ 

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

$$(\text{Suchzeit}(12) + \text{Suchzeit}(53) + \text{Suchzeit}(5) + \text{Suchzeit}(15) + \text{Suchzeit}(13) + \text{Suchzeit}(19)) / 6 = 9 / 6 = 1.5$$

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

- ▶ 

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

$$\text{Durchschnittliche Suchzeit} = 8 / 6 = 1.33\dots$$

# Algorithmus von Brent

## ▶ Beispiel

- ▶ 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	

## ▶ Aufwand

- ▶ Erfolgreiches Suchen:

$$1 + \frac{\alpha}{2} + \frac{\alpha^3}{4} + \frac{\alpha^4}{15} + \dots < 2,5$$

- ▶ Erfolgleses Suchen:  $\frac{1}{1-\alpha}$

```
// Hash-Tabelle t, Hash-Fkt. h, h_
void brentInsert (Comparable k)
{
    int i = h(k);
    while(t[i] != null)
    {
        int b = (i - h_(k)) % t.length;
        int bb = (i - h_(t[i])) % t.length;
        if(t[b] != null && t[bb] == null)
        {
            Comparable kk = k;
            k = t[i]; t[i] = kk;
            i = bb;
        }
        else
            i = b;
    }
    t[i] = k;
}
```

# Dynamische Hashtabellen

## ▶ Motivation

- ▶ Bisher war die Größe  $m$  der Hash-Tabelle immer konstant, d.h. sie kann nur eine feste Anzahl von Werten aufnehmen
- ▶ Oft kennt man die Maximalanzahl der einzufügenden Werte nicht, möchte aber nicht unnötig viel Speicher verschwenden

## ▶ Ansatz

- ▶ Die Tabelle wächst und schrumpft in Zweierpotenzen, so dass sie immer zu weniger als 50% gefüllt ist
- ▶ Vergrößerung und Verkleinerung sind teure Operationen, da alle Werte neu wieder eingefügt werden müssen

## ▶ Bessere Methoden

- ▶ Siehe Ottmann/Widmeyer

```
// Hash-Tabelle t, Füllstand n
void insert(Comparable k)
{
    int i = h(k);
    while(t[i] != null)
        i = (i + 1) % t.length;
    t[i] = k;
    if(++n > t.length / 2)
        expand();
}

private void expand()
{
    Comparable[] t2 = t;
    t = new Comparable[t2.length * 2];
    for(int i = 0; i < t2.length; ++i)
        if(t2[i] != null)
            insert(t2[i]);
}
```

