# Informatik II: Algorithmen und Datenstrukturen SS 2017

## Vorlesung 4b, Mittwoch, 17. Mai 2017
(Hash Maps, Rehash, Cuckoo Hashing)

Prof. Dr. Hannah Bast
Lehrstuhl für Algorithmen und Datenstrukturen
Institut für Informatik
Universität Freiburg

# Blick über die Vorlesung heute

- Drumherum
  - Placebo-Effekt — Einbildung?
- Inhalt
  - Hashtabellen — Prinzip + Beispiele
  - Dynamisches Hashing — Rehash
  - Cuckoo Hashing — Prinzip + Beispiel

# Placebo-Effekt 1/4

- Ist der Placebo-Effekt Einbildung?
  - "Meine Mama sagt immer: Einbildung ist auch ne Bildung"
  - "Das ist wie einen leeren aber mit blatt-xx beschrifteten Ordner committen: man bekommt trotzdem 0 Punkte"
  - "Der Placebo-Effekt ist in dem Moment real, wenn dadurch irgendetwas bewirkt wurde"
  - "Er ist nicht Einbildung, er basiert auf Einbildung"
  - "Wenn es durch die Anwendung eines Placebos zu einer tatsächlichen Besserung beim Patienten kommt, ist es irreführend, diese Wirkung als Einbildung zu bezeichnen"
  - "Placebo-Effekt ist Einbildung, Medikamente helfen wirklich"
  - "Sind wir Einbildungen?"

# Placebo-Effekt 2/4

- Eine bekannte und vielzitierte Studie von 2002
  - B. Moseley et al: A Controlled Trial of Arthroscopic Surgery for Osteoarthritis of the Knee, N Engl J Med 2002
  - Hintergrund: in den USA 650.000 Knie-OPs / Jahr wegen Knie-Arthrose und der damit einhergehenden Schmerzen
  - Doppelblinder Vergleich von drei Behandlungen (je 60 Pat.)

    **Lavage:** Spülung des Gelenkes

    **Débridement:** Wundreinigungs-OP

    **Placebo:** Simulation einer Wundreinigungs-OP inklusive Schnitte und Narben, aber ohne die eigentliche Maßnahme
  - Ergebnis: Funktionsfähigkeit und Schmerzlinderung waren in der Placebo-Gruppe zeitweise am besten !

# Placebo-Effekt 3/4

- Ist Placebo besser als Nicht-Behandlung?
  - Auch hierzu gibt es zahlreiche Studien
  - Die Ergebnislage ist gemischt

    Bei vielen Problemen (z.B. Depression, Bluthochdruck, Schlaflosigkeit) wird in der Regel kein Unterschied zwischen Placebo und Nicht-Behandlung gefunden

    Bei Schmerz-Problemen (z.B. Arthrose) aber teilweise deutliche Unterschiede bezüglich Schmerz **und** Funktion

    Interessant: im letzteren Fall sind Schnitte, Injektionen, Nadeln, etc. effektiver als Pillen
  - Ein Fazit daraus ist definitiv: anstatt sich darüber zu wundern, sollte es sich die Medizin zu Nutze machen!

# Placebo-Effekt 4/4

- Nutzen in der Medizin
  - Bei alternativen Heilmethoden heißt es oft abwertend: "Das ist doch nur Placebo"
  - Allerdings ist die positive Wirkung vieler klassischer Maßnahmen (Medikamente / OPs) auch "nur Placebo"

    Nur dass oft noch Nebenwirkungen dazu kommen
  - Ärzte geben oft Quasi-Placebos ("harmlose" Medikamente) wenn der Patient unbedingt eine Behandlung wünscht
  - Nicht-Intervention ist wohl bei einem Großteil aller Behand-lungsanliegen (natürlich nicht bei allen) die beste Therapie

    Voltaire: "Das Geheimnis der Medizin besteht darin, den Patienten abzulenken, während die Natur sich selber hilft"

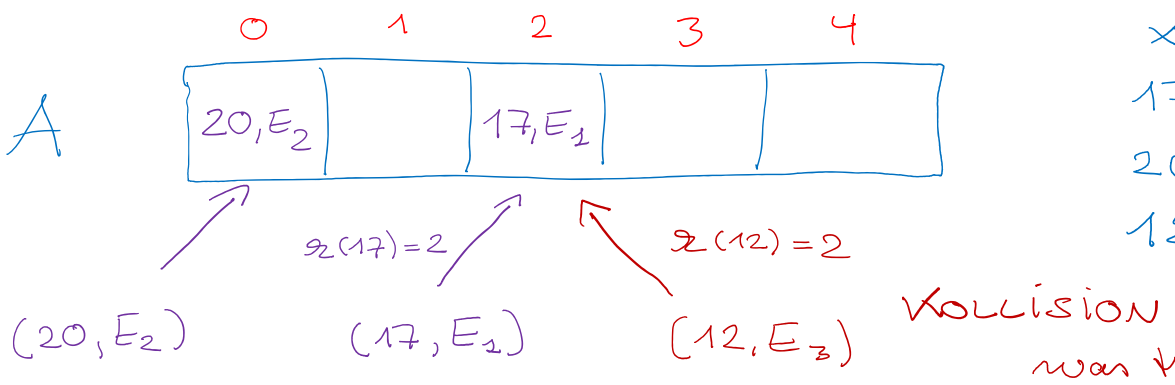
# Hash Map 1/8

Universum

- Grundidee
  - **Hash-Tabelle:** ein Feld **A** der Größe **m**
  - **Hash-Funktion:** eine Funktion **h** : U → {0, …, m – 1}

    GROß KLEIN
  - Speichere Element mit Schlüssel x unter A[h(x)]

    Problem: Kollisionen → $x1 \neq x2$ mit $h(x1) = h(x2)$
  - Beispiel mit m = 5 und h(x) = x mod 5
  - Wir fügen nacheinander ein: $(17, E_1)$ , $(20, E_2)$ , $(12, E_3)$




| x | h(x) |
|---|---|
| 17 | 2 |
| 20 | 0 |
| 12 | 2 |

KOLLISION

was tun?

# Hash Map 2/8

- Kollisionen, Lösung 1
  - Hashing mit **Verkettung**
  - Jeder Eintrag der Hashtabelle kann nicht nur ein key-value Paar speichern, sondern eine Menge davon

    Array<Array<KeyValuePair>> hashTable;

  - Beispiel mit $m = 5$ und $h(x) = x \bmod 5$ und Operationen:

$h(17) = 2$

insert(17, $E_1$)

$h(20) = 0$

insert(20, $E_2$)

$h(12) = 2$

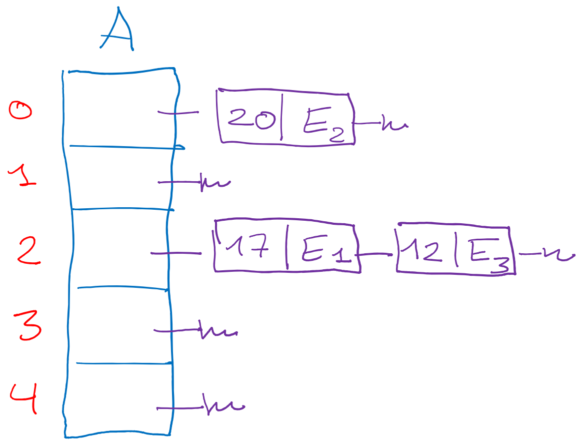
insert(12, $E_3$)

$h(x) = 2$

lookup(17) $\rightarrow E_1$

$h(x) = 2$

lookup(32) $\rightarrow$ nix



# Hash Map 3/8

z.B. $i \to (i+1) \bmod 5$
es genügt aber auch jede andere deterministische Formel

ERASE geht nicht so ohne Weiteres!

- Kollisionen, Lösung 2
  - Hashing mit **offener Adressierung**
  - Wenn eine Zelle schon besetzt ist, solange "ein Zelle weiter" gehen, bis man eine freie Zelle findet

    Man braucht dafür einen speziellen Schlüssel "Zelle ist frei" (NIX)
  - Beispiel mit $m = 5$ und $h(x) = x \bmod 5$ und Operationen:

    $h(x)=2$
    insert(17, $E_1$)
    $h(x)=0$
    insert(20, $E_2$)
    $h(x)=2$
    insert(12, $E_3$)
    $h(x)=2$
    A[2] anschauen — lookup(17) → $E_1$
    $h(x)=2$
    A[2] anschauen, A[3] anschauen, A[4] anschauen — lookup(32) → NIX
    $h(x)=3$
    insert(33, $E_4$)
    $h(x)=2$
    muss A[2], A[3], A[4], A[0], A[1] anschauen — lookup(32) → NIX

| | A |
|---|---|
| 0 | 20, $E_2$ |
| 1 | NIX |
| 2 | 17, $E_1$ |
| 3 | 12, $E_3$ |
| 4 | 33, $E_4$ |

Geht natürlich nur bis $\leq m$ Schlüssel

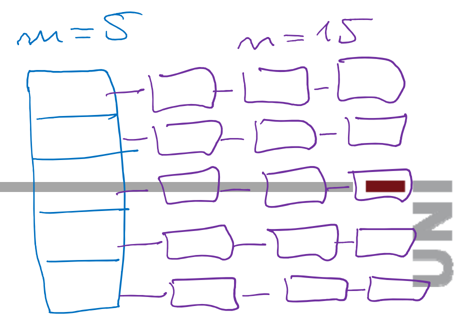
# Hash Map 4/8

- Vorgehen insert / lookup / erase
  - Im schlechtesten Fall muss man alle Elemente durchgehen, deren Schlüssel auf denselben Wert abgebildet werden

    Bei lookup kann man aufhören, wenn man den Schlüssel gefunden hat (mit Glück schon am Anfang)

    Bei Hashing mit Verkettung, kann man bei insert einfach am Ende anfügen

# Hash Map 5/8




- Laufzeit … bei Hashing mit Verkettung
  - **Best case:** die n Schlüssel werden von der Hashfunktion gleichmäßig verteilt

    Dann gehen insert und lookup in Zeit O(n / m)

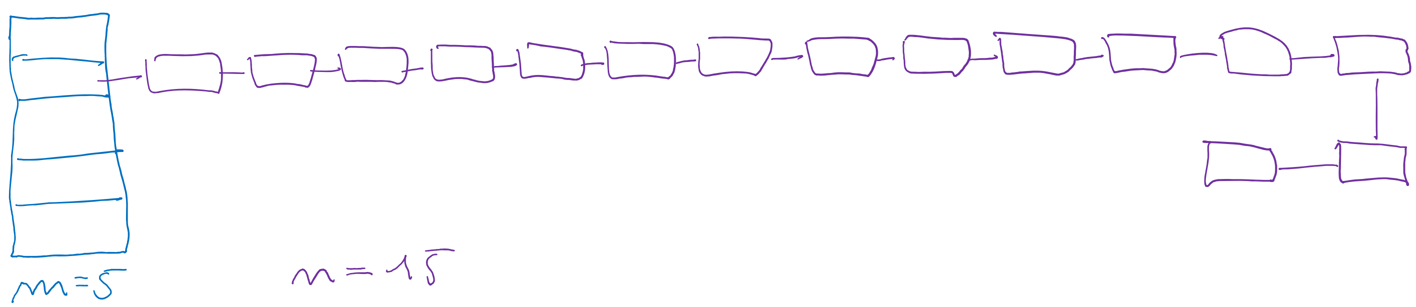
    Falls n = O(m) also in Zeit O(1)

    z.B. $n = 1.000.000$, $m = 100.000$ $\Rightarrow n/m = 10$

  - **Worst case:** alle n Schlüssel werden von der Hashfunktion auf denselben Wert abgebildet

    Dann braucht lookup im schlechtesten Fall Θ(n)

    Wie bei Realisierung 2 (Folie 15 von Vorlesung 4a gestern)



# Hash Map 6/8

- Wahl der Hashfunktion, zufällige Schlüssel
  - Bei zufällig verteilten Schlüsseln gibt die einfache Funktion $h(x) = x \bmod m$ schon die bestmögliche Verteilung

    **Intuitiv:** für zufälliges $x$ ist auch $x \bmod m$ zufällig aus $\{0, \ldots, m - 1\}$, und so bekommt jede Zelle der Hashtabelle im Erwartungswert gleich viele Schlüssel (und zwar $n / m$)

    Das machen wir nächste Woche genauer

    Ich gebe Ihnen da auch einen kleinen Auffrischungskurs in Wahrscheinlichkeitsrechnung (**#endlichwiedermathe**)

# Hash Map 7/8

- Wahl der Hashfunktion, nicht-zufällige Schlüssel
  - Bei nicht-zufällig verteilten Schlüsseln kann die Hashfunktion h(x) = x mod m beliebig schlecht sein
  - Beispiel: m = 10 und Schlüssel 21, 11, 51, 71, 61, …
  - Was man dagegen macht, sehen wir nächste Woche

    Für das ÜB4 können Sie trotzdem einfach h(x) = x mod m verwenden … und feste daran glauben, dass es klappt

# Hash Map 8/8

$h(\text{"dooof"}) =$
$(100 + 111 + 111 + 102) \bmod 5$
$= 4$

$m = 5$

- Schlüssel, die keine Zahlen sind
  - **Option 1:** Jedes im Rechner repräsentierte Objekt kann als Zahl aufgefasst werden, zum Beispiel

    Sei Objekt in $k$ Bytes $B_0 \ldots B_{k-1}$ repräsentiert, dann entspricht der Inhalt dieser Bytes eindeutig der Zahl $\Sigma_{j=0,\ldots,k-1} B_j \cdot 256^j$
  - **Option 2:** Objekt direkt auf $\{0, \ldots, m - 1\}$ abbilden (= "hashen"), ohne Umweg über eine Zahl, z.B. für string $s$

    $h(s)$ = Summe der ASCII-Codes der Zeichen $\bmod m$

    Das können Sie zum Beispiel für das ÜB4 verwenden

    Aber wieder keine Garantie, dass es gut verteilt ist, die gibt es erst ab nächster Woche

# Rehash 1/4

insbesondere die Anzahl |S|

- Bisherige Annahme
  - Die Schlüsselmenge S ist vorher bekannt

    bei Verkettung ; bei off. Addr. $m \geq 2n$ oder so
  - Dann kann man leicht die Größe der Hashtabelle als $m = \Theta(n)$ wählen, so dass die Anzahl Schlüssel, die auf denselben Wert abgebildet werden im besten Fall $\Theta(1)$ ist

    Dann gehen auch insert und lookup in Zeit $\Theta(1)$
  - Es können aber zwei Dinge passieren

    Es kommen Schlüssel dazu (und wir wissen vorher nicht, wie viele) und die Hashtabelle wird zu klein

    Wir haben Pech und es werden übermäßig viele Schlüssel auf denselben Wert abgebildet

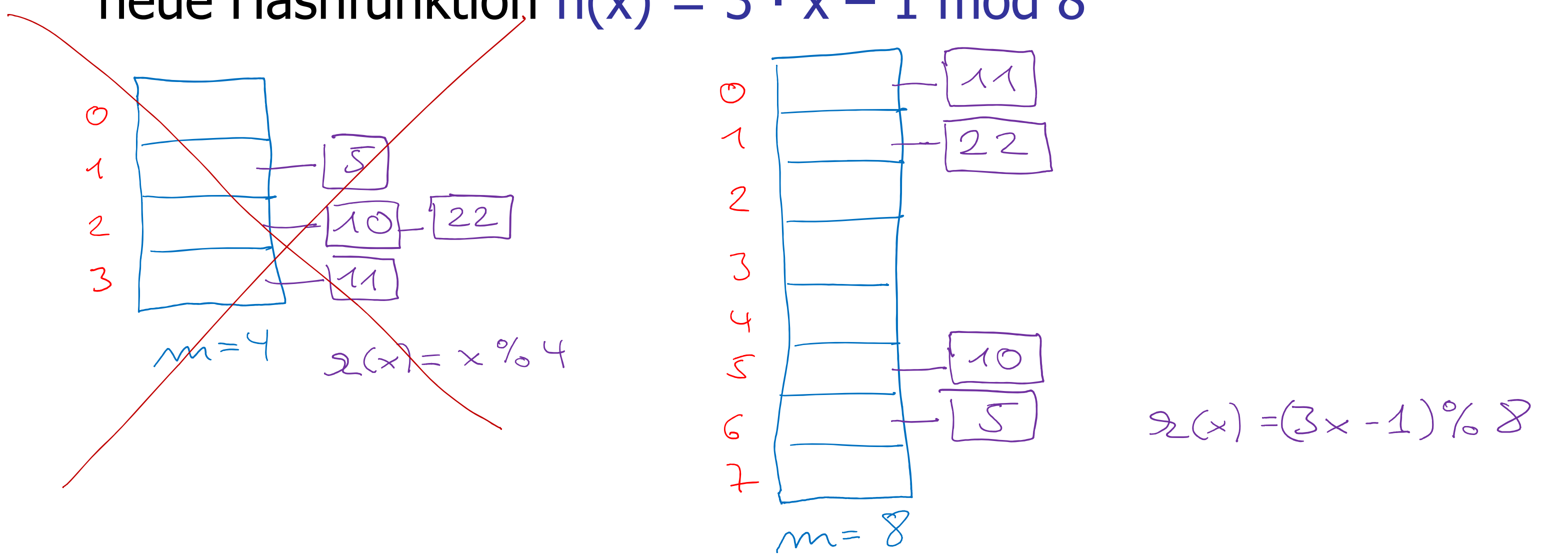
    Das Gute daran: beides kann man leicht feststellen

# Rehash 2/4

in der neuen Tabelle, neue H-Funktion verwenden

Elemente lassen wir der Einfachheit halber zu mal weg

- Lösung für beide Probleme: **Rehash**
  - Bei einem Rehash macht man einfach Folgendes:
    1. Eine neue Hashfunktion auswählen
    2. Die Elemente von der alten in die neue Tabelle kopieren
    3. Die alte Tabelle löschen
  - **Beispiel:** $S = \{5, 10, 11, 22\}$, alte Fkt $h(x) = x \bmod 4$, neue Hashfunktion $h(x) = 3 \cdot x - 1 \bmod 8$



# Rehash 3/4

- Kosten für einen Rehash
  - Ein Rehash ist teuer: er kostet Zeit $\Theta(n)$, wobei $n$ die Anzahl Elemente zum Zeitpunkt des Rehash ist
  - Wenn man es richtig macht, ist er allerdings selten nötig:

    Mit clever gewählten Hashfunktionen (siehe nächste Woche) ist das unwahrscheinlich

    Wenn die Hashtabelle zu klein geworden ist, und man die neue Hashtabelle doppelt so groß wählt ($\mathbf{m} \rightarrow \mathbf{2m}$), dauert es lange, bis man wieder vergrößern muss

    Diese "Verdoppelungsstrategie" analysieren wir in Vorlesung 6a und 6b genauer (amortisierte Analyse)

# Rehash 4/4

- Verkleinerung der Schlüsselmenge
  - Die Schlüsselmenge kann auch wieder kleiner werden, indem Schlüssel gelöscht werden

    Python: **del**     Java: **remove**     C++: **erase**
  - Wenn $|S| \ll m$ wird, kann man die Hashtabelle auch wieder verkleinern ... siehe ebenfalls Vorlesung 6a+b
  - Macht man aber in der Praxis oft nicht, weil:

    1. In sehr vielen Anwendungen braucht man nur **insert** und **lookup**, kein del / remove / erase
    2. Zu irgendeinem Zeitpunkt braucht man sowie den Platz für die maximale Anzahl Schlüssel der Anwendung

# Cuckoo Hashing 1/5

- Beschreibung des Algorithmus
  - Es gibt eine Hashtabelle der Größe $m$ wie gehabt
  - Es gibt **zwei** Hashfunktionen $h_1, h_2 : U \rightarrow \{0, \ldots, m-1\}$
  - Jede Position der Hashtabelle hat nur Platz für **ein** Element
  - Versuche ein neues Element $x$ bei $h_1(x)$ zu speichern
  - Falls schon belegt von einem Element $y$, dann speichere $y$ bei $h_i(y)$ falls vorher bei $h_j(y)$ gespeichert, $\{i, j\} = \{1, 2\}$

    Für jeden Schlüssel gibt es also **genau** zwei mögliche Plätze
  - Falls neuer Platz für $y$ belegt von einem Element $z$: dann verfahre genau so mit $z$ … und so weiter

# Cuckoo Hashing 2/5

- Zyklus
  - Es kann so zu einem **Zyklus** kommen, und zwar wenn für eine Teilmenge von Schlüsseln $S'$ gilt:

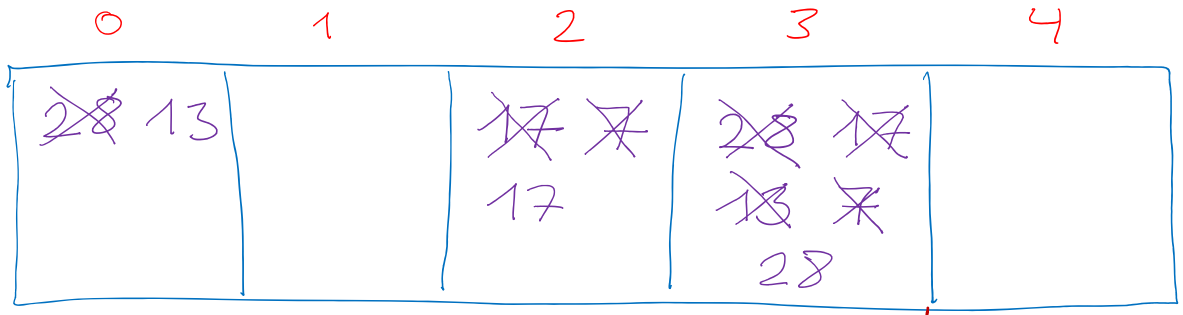
    $|\{h_1(x) : x \in S'\} \cup \{h_2(x) : x \in S'\}| < |S'|$

    Intuitiv: es gibt weniger Plätze als Schlüssel
  - Wenn das passiert, wählt man neue Hashfunktionen $h_1$ und $h_2$ und macht einen **Rehash** wie erklärt

# Cuckoo Hashing 3/5

| $x$ | $h_1(x)$ | $h_2(x)$ |
|---|---|---|
| 17 | 2 | 3 |
| 28 | 3 | 0 |
| 7 | 2 | 3 |
| 13 | 3 | 0 |

- Beispiel ... ohne Rehash, nur mit Schlüsseln, ohne Elemente
  - Mit: $m = 5$, $h_1(x) = x \bmod 5$, $h_2(x) = 2x - 1 \bmod 5$
  - Füge nacheinander ein: 17, 28, 7, 13




ZYKLUS!

weil für $\underbrace{17, 28, 7, 13}_{\text{4 Schlüssel}}$

nur Plätze $\underbrace{0, 2, 3}_{\text{3 Plätze}}$

# Cuckoo Hashing 4/5

- Wahl der Hashfunktionen
  - Sollte man **unabhängig** voneinander wählen, so dass jede für sich eine gute Hashfunktion wäre

    D.h. die Schlüssel werden möglichst gleichmäßig verteilt

    Dazu mehr nächste Woche, wie man das hinkriegt
  - Dann kann man zeigen, dass es hinreichend selten zu einem Zyklus (und dem dann nötigen teuren Rehash) kommt, solange $|S| \leq m/2$
  - Bei wachsender Schlüsselmenge wie gehabt ein Rehash mit $\mathbf{m} \rightarrow \mathbf{2m}$, zum Beispiel sobald $|S| > m/2$

# Cuckoo Hashing 5/5

- Laufzeit
  - Die Laufzeit von lookup(x) ist **immer** $\Theta(1)$

    Man muss ja immer nur an zwei Positionen nachschauen, nämlich $h_1(x)$ und $h_2(x)$ … **das ist gerade der Clou**
  - Dasselbe gilt dann auch für remove(x)

    An beiden Positionen nachschauen, und wo gefunden einfach löschen, die Position ist dann wieder frei
  - Man kann zeigen, dass ein insert(x) **im Durchschnitt** in Zeit $\Theta(1)$ geht

    Beweis siehe Referenzen … aber nicht Klausur-Elefant

# Literatur / Links

- Universelles Hashing
  - In Mehlhorn / Sanders:

    4 Hash Tables and Associative Arrays
  - In Wikipedia

    http://en.wikipedia.org/wiki/Hash_table
- Cuckoo Hashing
  - http://en.wikipedia.org/wiki/Cuckoo_hashing