Informatik II: Algorithmen und Datenstrukturen SS 2015

Vorlesung 4b, Mittwoch, 13. Mai 2015

(HashMaps, Cuckoo Hashing, Rehash)

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

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Tempo der Vorlesung
- Hashtabellen
  - Grundprinzip: Hashtabelle, Hashfunktion, Kollisionen
  - Dynamisches Hashing: Rehash
  - Cuckoo Hashing: praktisch und elegant

  Ü4, Aufgabe 1: Schreiben Sie Ihre eigene Klasse HashMap (mit einer Hashmethode Ihrer Wahl, ohne Rehash)

  Ü4, Aufgabe 2: Finden Sie die häufigste Suchanfrage, mit Ihrer eigenen Klasse aus Aufgabe 1

Tempo der Vorlesung

UNI FREIBURG

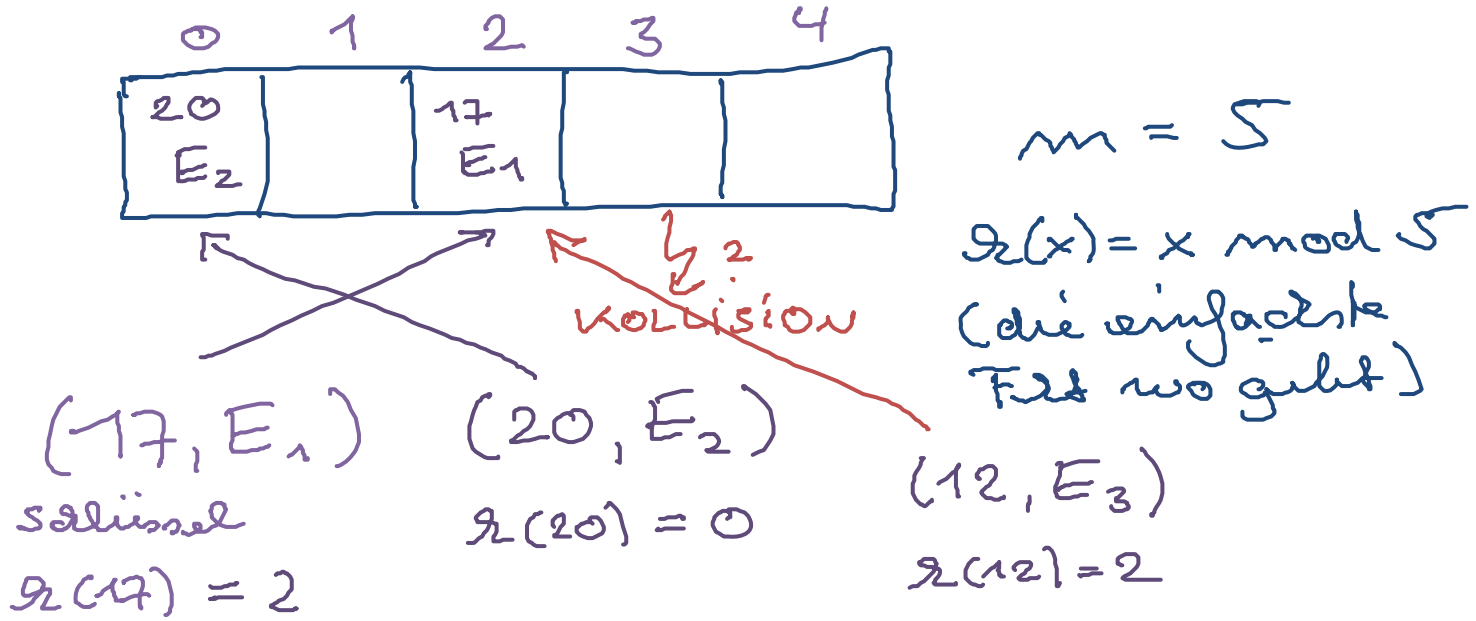
- Bitte um Feedback
  - Es ist ja immer so, dass es einigen zu schnell geht und einigen zu langsam

    Vorwissen, Kapazität und persönliches Lerntempo sind einfach sehr unterschiedlich
  - Meine Aufgabe ist es, einen guten Mittelweg zu finden
  - Geben Sie mir deshalb in den nächsten erfahrungen.txt bitte Feedback dazu (siehe letzte Frage auf dem Ü4)

    Tipp: wenn es Ihnen zu langsam geht, machen Sie doch während der Vorlesung schon mal das Übungsblatt

HashMap 1/8

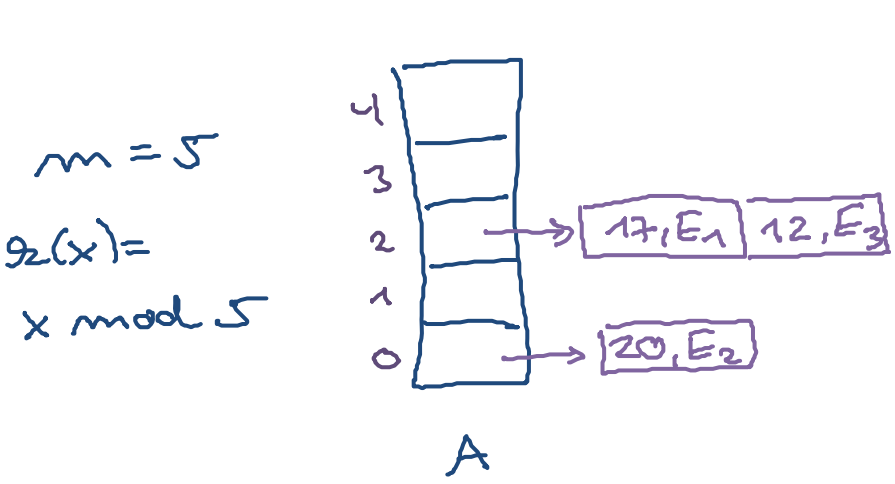
- Grundidee
  - **Hash-Tabelle:** ein Feld **A** der Größe **m**
  - **Hash-Funktion:** eine Funktion **h** : U → {0, …, m – 1}
  - Speichere Element mit Schlüssel x unter A[h(x)]
- Beispiel



HashMap 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;



insert (17, E_1)
insert (20, E_2)
insert (12, E_3)
lookup (17) → 17, E_1
lookup (32) → Nix

HashMap 3/8

- Kollisionen, Lösung 2
  - Hashing mit **offener Adressierung**
  - Wenn eine Zelle schon besetzt ist, solange eins nach rechts gehen, bis man eine freie Zelle findet

    Dafür braucht man einen speziellen Wert, der bedeutet "Zelle ist frei", der sonst nicht vorkommt

wenn man bei m-1 ankommt, wieder bei 0 anfangen

NIX

$m = 5$, $h(x) = x \bmod 5$

0	1	2	3	4
20 E_2	NIX	17 E_1	12 E_3	33 NIX E_4

insert(17, E_1)
insert(20, E_2)
insert(12, E_3)
lookup(17) ✓
lookup(32) ×
insert(33, E_4)
lookup(32) ×

Löschen geht hier nicht ohne Weiteres

für lookup(32) kann man hier aufhören.

HashMap 4/8

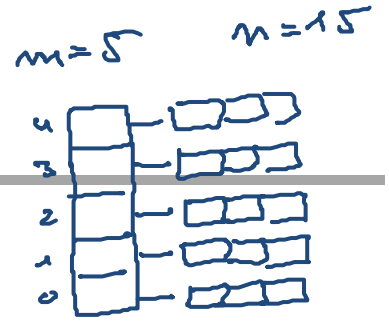
UNI FREIBURG

- 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

HashMap 5/8



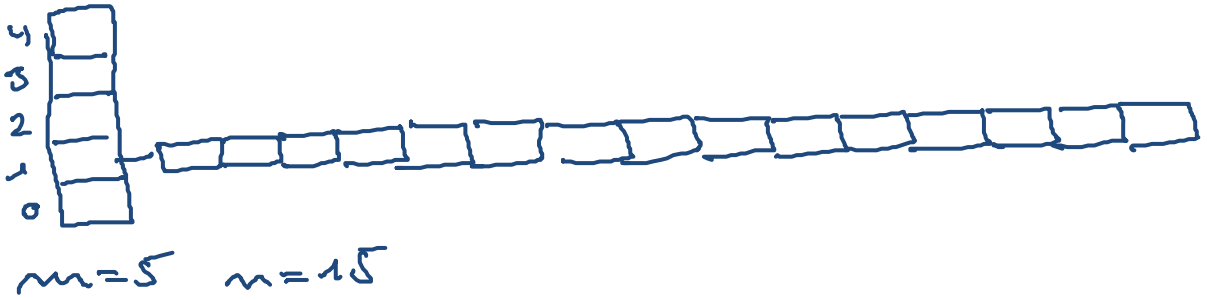
- Laufzeit … bei Hashing mit Verkettung
  - **Best case:** die n Schlüssel werden von der Hashfunktion gleichmäßig verteilt

    Dann gehen insert und lookup in Zeit O(n / m) statt $\Theta(m)$

    Falls n = O(m) also in Zeit O(1)
  - **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 8 von Vorlesung 4a)



HashMap 6/8

UNI FREIBURG

- 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 (noch besser als Integrale)

HashMap 7/8

UNI FREIBURG

- Wahl der Hashfunktion, nicht-zufällige Schlüssel
  - Bei nicht-zufällig verteilten Schlüsseln kann die Hashfunktion $h(x) = x \bmod m$ beliebig schlecht sein
  - Beispiel: $m = 10$ und Schlüssel 21, 11, 51, 71, 61, …

    Was man dagegen macht, sehen wir nächste Woche

    Für das Ü4 können Sie einfach $h(x) = x \bmod m$ verwenden und beten*, dass es klappt

    *Alternativ: Niederwerfungen, Rosenkranz, kalte Dusche, …

HashMap 8/8

x : d o o f m = 5
100 111 111 102 ← ASCII codes
g(x) = (100 + 111 + 111 + 102) mod 5
= 4

- 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 mod m

    Können Sie zum Beispiel für das Ü4 verwenden

    Aber wieder keine Garantie, dass es gut verteilt

Rehash 1/4

z.B $n = 1.000.000$
$m = 100.000$
$\Rightarrow \frac{n}{m} = 10$

- Bisherige Annahme
  - Die Schlüsselmenge S ist vorher bekannt, insbes. $n = |S|$
  - 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 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

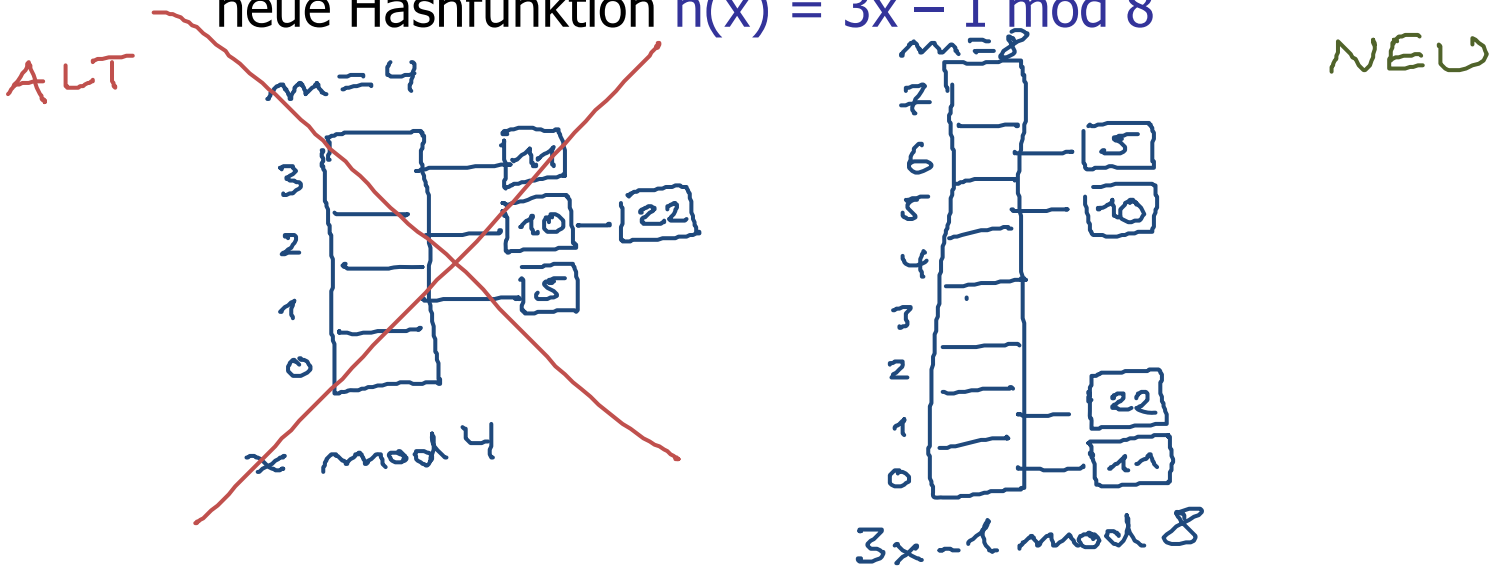
    Gut daran: beides kann man leicht feststellen

Rehash 2/4

- Lösung für beide Probleme: **Rehash**
  - Bei einem Rehash wird einfach:
    1. Eine neue Hashfunktion ausgewählt
    2. Die Elemente von der alten in die neue Tabelle kopiert
    3. Die alte Tabelle gelöscht

der Einfach halber ohne "values" / Elemente

  - **Beispiel:** S = {5, 10, 11, 22}, alte Fkt h(x) = x mod 4, neue Hashfunktion h(x) = 3x – 1 mod 8



Rehash 3/4

UNI FREIBURG

- 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 ($\mathbf{m \rightarrow 2m}$), dauert es lange, bis man wieder vergrößern muss

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

Rehash 4/4

UNI FREIBURG

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

    Python: del Java: remove C++: erase
  - Wenn |S| << 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

UNI FREIBURG

- Beschreibung des Algorithmus
  - Es gibt eine Hashtabelle der Größe m wie gehabt
  - Es gibt **zwei** Hashfunktionen $h_1, h_2 : U \to \{0, \ldots, m-1\}$
  - 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, $\{i, j\} = \{1, 2\}$
  - Falls neuer Platz für y belegt von einem Element z, dann verfahre genau so mit z … und so weiter

Cuckoo Hashing 2/5

UNI FREIBURG

- 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

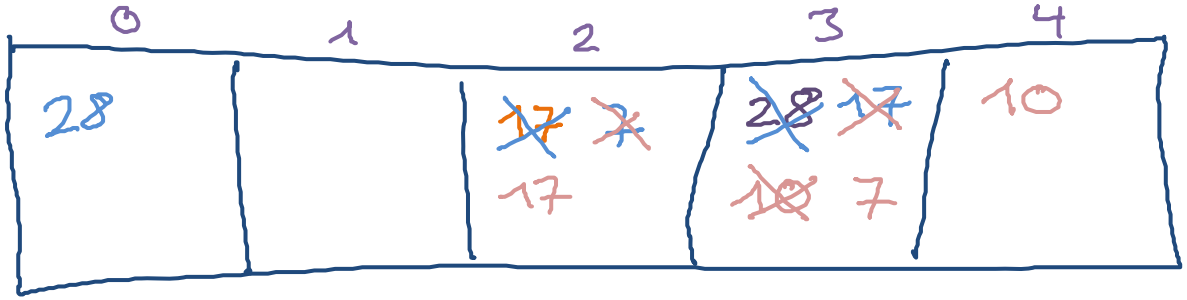
Cuckoo Hashing 3/5

x	$g_1(x)$	$g_2(x)$
17	2	3
28	3	0
7	2	3
10	3	4

$$g_1(x) = \begin{cases} x \bmod 5 & x \neq 10 \\ 3 & x = 10 \end{cases}$$

- Beispiel (ohne Rehash) ... insert: 17, 28, 7, 10, 20

m = 5, ~~$h_1(x) = x \bmod 5$~~, $h_2(x) = 2x - 1 \bmod 5$



insert (17): 17

insert (28): 28

insert (7): 7 → 17 → 28

insert (10): 10 → 17 → 7 → 10

insert (20): ZYKLUS

Cuckoo Hashing 4/5

UNI FREIBURG

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

    D.h. die Schlüssel 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 $\mathbf{m \rightarrow 2m}$, zum Beispiel sobald $|S| > m/2$

Cuckoo Hashing 5/5

UNI FREIBURG

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

    auch worst case nicht nur „average case"

    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) bzw. erase

    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