universität freiburg

# Algorithmen und Datenstrukturen SS 2025

## Vorlesung 5: Assoziative Arrays, Hash Maps

**Dienstag, 20. Mai 2025**

Prof. Dr. Hannah Bast
Professur für Algorithmen und Datenstrukturen
Institut für Informatik
Albert-Ludwigs-Universität Freiburg

# Blick über die Vorlesung heute

- Organisatorisches
  - Erfahrungen mit dem Ü4 — O-Notation
  - Infos zur Klausur — Termin steht jetzt fest
  - Vorgucker auf das Ü5 — Hash Map coden und anwenden
- Inhalt
  - Assoziative Felder — Definition, Motivation, Beispiele
  - Hash Maps — Grundidee, Varianten, Laufzeit
  - Rehash — bei dynamischer Schlüsselmenge
  - Bibliotheken dafür — in Python / Java / C++

# Erfahrungen mit dem Ü4 1/2

- Auszüge aus Ihrem Feedback [halbwegs repräsentativ]

"Blatt viel Spaß gemacht, Vorlesung sehr hilfreich und interessant"

"Das Blatt war die richtige Kombo aus spaßig und knobelig"

"Gut machbar, musste nochmal die Logarithmusgesetze lernen"

"Aufgaben waren simpel, aber nicht leicht zu bearbeiten"

"V4 war eine nette Abwechslung, nachdem V3 ziemlich schwer war"

"Vielen Dank für das Feedback, genau so hilft mir das weiter"

"Bin mir mit der Stichhaltigkeit meiner Beweise sehr unsicher"

"Ich vermute, ich habe die Aufgabe 2 viel zu kompliziert gemacht"

"Blatt fand ich nicht so sinnvoll, muss man wirklich viel denken"

"Leider gemerkt, dass mir die mathematischen Grundlagen fehlen"

# Erfahrungen mit dem Ü4 2/2

- Sorge, dass zukünftiger Job von KI übernommen wird?
  - "KI wird Teilaufgaben übernehmen, aber nicht die gesamte Arbeit"
  - "Ein bisschen, aber wird noch dauern bis KI unseren Skill erreicht"
  - "Zumindest nicht bis es eine richtige KI ist und nicht nur ein LLM"
  - "Durch Copilot wird Arbeit nicht weggenommen sondern effizienter"
  - "Informatiker braucht man immer und ist die Sprache der Zukunft"
  - "Informatiker generell wahrscheinlich nicht, ich aber schon"
  - "Ein Zimmerer wird nicht durch einen automatischen Hammer ersetzt"
  - "Ich studiere Lehramt, da kommt es aufs Zwischenmenschliche an"
  - "Mache gezielt ESE, um in Zeiten von KI breiter aufgestellt zu sein"
  - "Angst vor KI? I am the one who knocks!"

# Termine für die Klausur

- Stehen jetzt fest
  - **Klausur**

    Freitag **8. August** von 14:00 – 16:xx h in Gebäude 101, TF

    Genauere Infos dazu in der letzten Vorlesung
  - **Korrektur**

    Am 15. + 16. August … eigentlich korrigieren wir immer am selben Tag noch, aber das geht diesmal leider nicht aufgrund des Klausurtermins (den wir nicht selber festgelegt haben)
  - **Einsicht**

    In der Woche 18. – 22. August … genauere Info zu Zeit, Ort und Modalitäten finden Sie rechtzeitig auf dem Wiki

# Vorgucker auf das Ü5

- Praktisches Blatt mit interessantem (echten) Datensatz

  **A1** Implementieren Sie eine Hash Map

  Schöne Aufgabe, um das umzusetzen und zu vertiefen, was Sie in der Vorlesung heute (hoffentlich) gelernt haben

  **A2** Anwendung von A1 auf besagtem Datensatz

  Die Noten aller Prüfungsleistungen an der TF der letzten 10 Jahre, inklusive Namen, Adressen und Matrikelnummern

  Nee, die Daten sind natürlich anonymisert und auch leicht perturbiert … sie stimmen aber im Prinzip

  Sie sollen mit einer Hash Map die Durchschnittsnoten und Durchfallquoten pro Lehrveranstaltung und Semester berechnen

  Erzählen Sie uns, was Sie für Trends entdecken

# Assoziative Felder aka Maps 1/8

- Definition
  - Verwalten einer Menge von $n$ Elementen, jedes mit einem eindeutigen Schlüssel $S$ aus einer beliebigen Menge $U$

    Schlüssel = **Key**, Wert = **Value** (beliebige Daten)
  - Uns interessieren dabei insbesondere die folgende Operationen:

    insert(key, value) Einfügen von Wert value mit Schlüssel key

    lookup(key) Ist key $\in$ S und wenn ja mit welchem value

    erase(key) Falls key $\in$ S, dann das Elemente löschen

    Wir schauen uns heute (fast) nur "insert" und "lookup" an
  - Terminologie: ein assoziatives Feld heißt in den gängigen Programmiersprachen meistens **Map** oder **Dictionary**

# Assoziative Felder aka Maps 2/8

- Anwendungsbeispiel 1
  - Matrikelnummern (Schlüssel) und Klausurergebnisse (Wert)

    | | |
    |---|---|
    | 3126932 | 1,0 |
    | 5148948 | 2,3 |
    | 3904433 | 3,7 |

  - Universum U = Menge aller möglichen Matrikelnummern

    Bei bis zu 7-stelligen Nummern: $U = \{0, \ldots, 9\,999\,999\}$

  - Man beachte: die Matrikelnummern sind zwar Zahlen, aber nicht unbedingt fortlaufend, zum Beispiel:

    Für eine bestimmte Klausur ist die Schlüsselmenge S (Matrikelnummern der Prüflinge) **viel** kleiner als das Universum U (alle möglichen Matrikelnummern)

# Assoziative Felder aka Maps 3/8

- Anwendungsbeispiel 2
  - Häufigkeiten von Vorkommen von Strings in einer Datei

    | | |
    |---|---|
    | Technische Informatik | 1831 |
    | Stochastik | 820 |
    | Softwaretechnik | 504 |

  - Jetzt sind die Schlüssel Zeichenketten und die Werte Zahlen

    Das Universum U sind in dem Fall die Menge aller möglichen Zeichenketten (und wir sehen nur einen Bruchteil davon)

    Für Aufgabe 2 vom Ü5 müssen Sie auch Vorkommen von Strings zählen, ähnlich (aber nicht genau so) wie oben, siehe dazu auch die Hinweise auf Folie 22

# Assoziative Felder aka Maps 4/8

- Anwendungsbeispiel 3
  - Die verschiedenen Zahlen aus der Eingabe zu einem Sortieralgorithmus, und wie oft sie jeweils vorkommen

    | | |
    |---|---|
    | 12 | 3 mal |
    | 17 | 2 mal |
    | 4 | 5 mal |

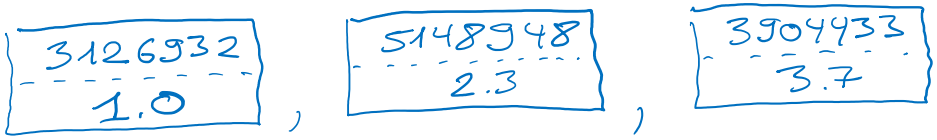
  - Das Universum U sind jetzt die Menge aller möglicher Zahlen und für eine bestimmte Eingabe ist die Schlüsselmenge S wieder nur eine kleine Teilmenge davon
  - Damit kann man CountingSort verallgemeinern auf Eingabezahlen **beliebiger Größe** …heißt dann **MapCountingSort**

    Falls O(n / log n) verschiedene Zahlen, sogar Laufzeit O(n)

    Mehr Details dazu in Vorlesung 4a der Info II vom SS 2017

# Assoziative Felder aka Maps 5/8

- Realisierung, Idee 1
  - Die |S| Key-Value Paare stehen in einer "Liste", insert hängt an das Ende der Liste an, lookup durchsucht das Feld von links nach rechts
  - Zum Beispiel: insert(3126932, "1.0"), insert(5148948, "2.3"), insert(3904433, "3.7"), lookup(1234567)




  - Dann bekommen wir:

wenn wir annehmen, dass man in O(1) an eine Liste anhängen kann

✓ Laufzeit insert: $O(1)$

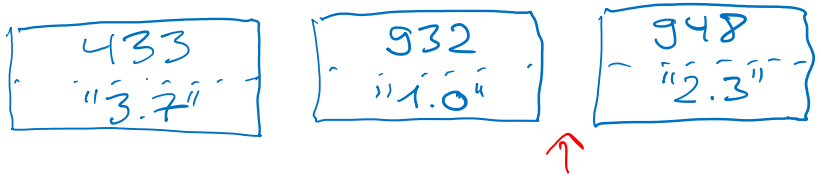
✗ Laufzeit lookup: bis zu $\Theta(|S|)$

✓ Platzverbrauch: $\Theta(|S|)$

# Assoziative Felder aka Maps 6/8

- Realisierung, Idee 2


  - Wie Idee 1, aber mit einem immer sortierten Feld ... dann geht lookup schneller (mit binärer Suche), aber insert langsamer (die Sortierung muss gewahrt bleiben)
  - Zum Beispiel: insert(932, "1.0"), insert(948, "2.3"), insert(433, "3.7"), lookup(615)




Zustand nach den drei inserts

  - Dann bekommen wir:

✗ Laufzeit insert: bis zu $\Theta(|S|)$

(✓) Laufzeit lookup: bis zu $\Theta(\log |S|)$

✓ Platzverbrauch: $\Theta(|S|)$

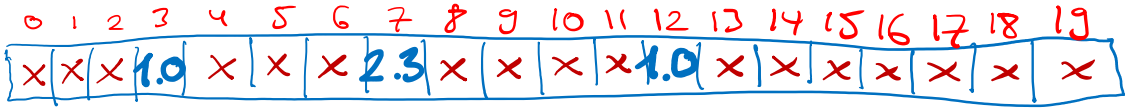
Finden, wo das neue Element reinpasst, dann einmal kopieren ...

ABER siehe Suchbäume in V8

# Assoziative Felder aka Maps 7/8

- Realisierung, Idee 3
  - Feld A der Größe |U|, für jeden möglichen Key i steht an A[i] das zugehörige Value, sonst ein spezielles NO_VALUE (und zu Beginn steht überall NO_VALUE)
  - Zum Beispiel für ein kleines Universum U = {0, ..., 19} :

    insert(3, "1.0"), insert(12, "1.0"), insert(7, "2.3"), lookup(5)




  - Dann bekommen wir:

✓ Laufzeit insert: $O(1)$

✓ Laufzeit lookup: $O(1)$

✗ Platzverbrauch: $\Theta(|U|)$ ... nicht $\Theta(|S|)$

# Assoziative Felder aka Maps 8/8

- Realisierung 4, Wunsch
  - Wir hätten gerne eine Datenstruktur, so dass für eine Schlüsselmenge $S$ aus einem beliebigen Universum $U$:

    Laufzeit insert: $\Theta(1)$

    Laufzeit lookup: $\Theta(1)$

    Platzverbrauch: $\Theta(|S|)$ ... nicht $\Theta(|U|)$
  - Das wäre das Optimum, denn besser geht es nicht
  - Mit einer sogenannten **Hash Map** geht das tatsächlich (fast)

    Das Prinzip sehen wir gleich, Aufgabe 1 des Ü5 ist dann, eine einfache Hash Map selber zu implementieren

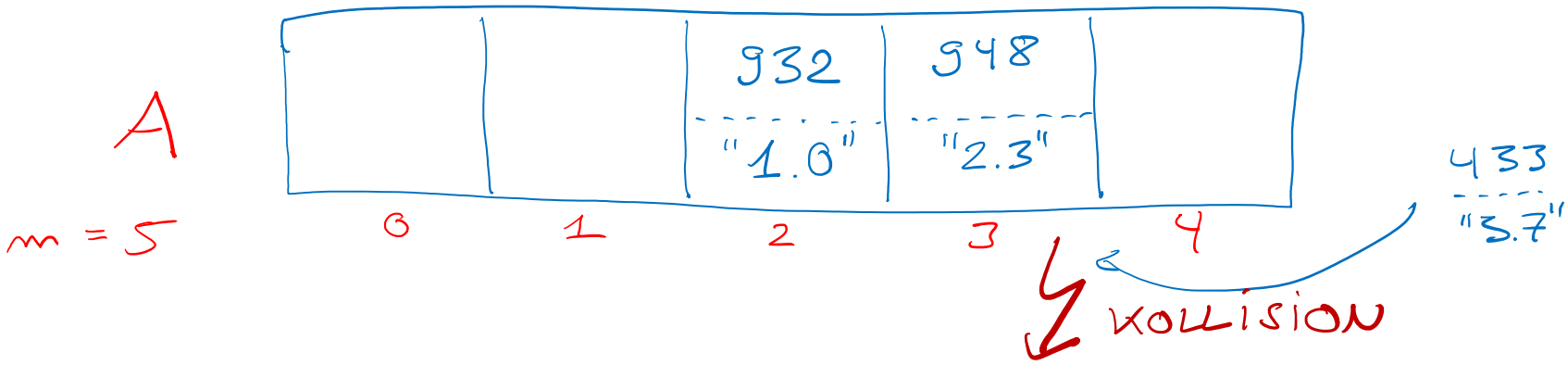
# Hash Map 1/8

- 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)]

    Problem: Kollisionen → $x_1 \neq x_2$ mit $h(x_1) = h(x_2)$
  - Beispiel mit m = 5 und h(x) = x mod 5 und den Operationen

    h(932) = 2 h(948) = 3 h(433) = 3

    insert(932, "1.0"), insert(948, "2.3"), insert(433, "3.7")

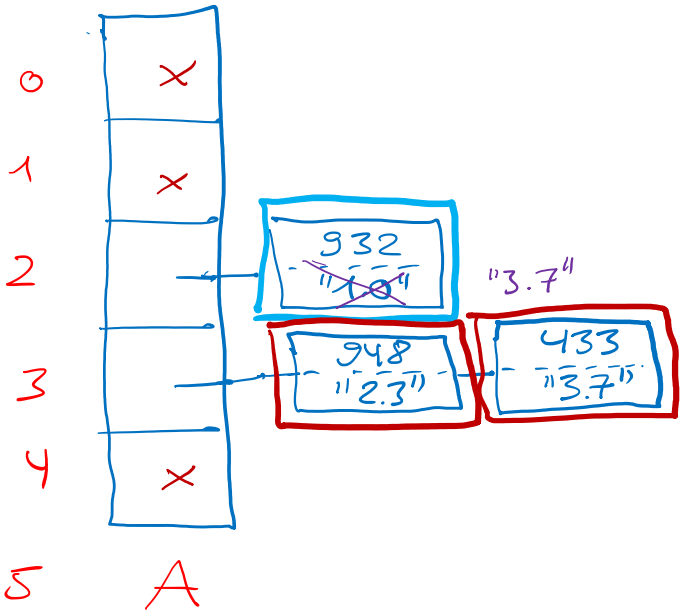


# Hash Map 2/8

- Kollisionen, Lösung 1
  - Hashing mit **Verkettung**
  - Jeder Eintrag der Hashtabelle kann nicht nur ein key-value Paar speichern, sondern eine Menge davon
  - Beispiel mit m = 5 und h(x) = x mod 5 und Operationen:

2 insert(932, "1.0")
3 insert(948, "2.3")
3 insert(433, "3.7")
2 lookup(932) → GEFUNDEN
3 lookup(328) → NICHT GEFUNDEN

insert (932, "3.7")



# Hash Map 3/8

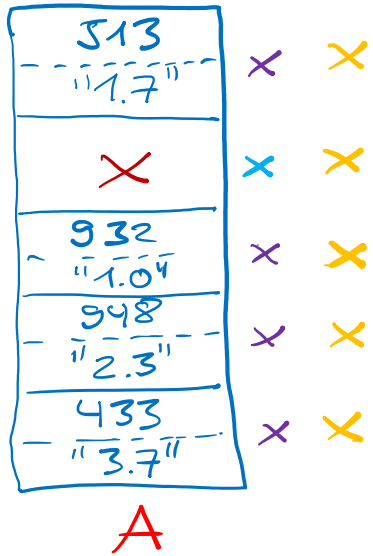
ACHTUNG: Funktioniert NICHT ohne Weiteres mit ERASE

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

    Man braucht wieder einen speziellen Wert NO_VALUE
  - Beispiel mit m = 5 und h(x) = x mod 5 und Operationen:

x = NO_VALUE

| | |
|---|---|
| 2 | insert(932, "1.0") |
| 3 | insert(948, "2.3") |
| 3 | insert(433, "3.7") |
| 1 | lookup(351) → NICHT GEFUNDEN |
| 2 | lookup(142) → NICHT GEFUNDEN |
| 3 | insert(513, "1.7") |
| 2 | lookup(142) → NICHT GEFUNDEN |



# Hash Map 4/8

- Details zu **insert**(key, value)

  ansonsten nennt man das: MULTI-Map

  - Hashing mit Verkettung: falls es in der Liste unter h(key) schon ein Element gibt mit demselben key, wird der Wert einfach überschrieben (und der alte Wert damit gelöscht)
  - Hashing mit offener Adressierung: kann maximal m Elemente speichern, wenn m die Größe der Hashtabelle ist

- Details zu **lookup**(key)

  - Sobald man ein Element mit dem key gefunden hat → fertig
  - Wenn es kein Element mit dem key gibt, muss man bei

    … Verkettung: alle Elemente mit demselben h(key) durchgehen

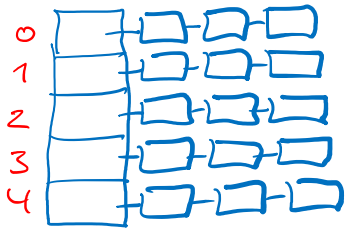
    … offener Adressierung: so lange durch die Tabelle gehen, bis man auf eine Zelle mit dem Wert NO_VALUE trifft

# Hash Map 5/8

- Laufzeit bei Hashing mit Verkettung
  - **Bester Fall ("best case"):** die n Schlüssel werden von der Hashfunktion gleichmäßig verteilt

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

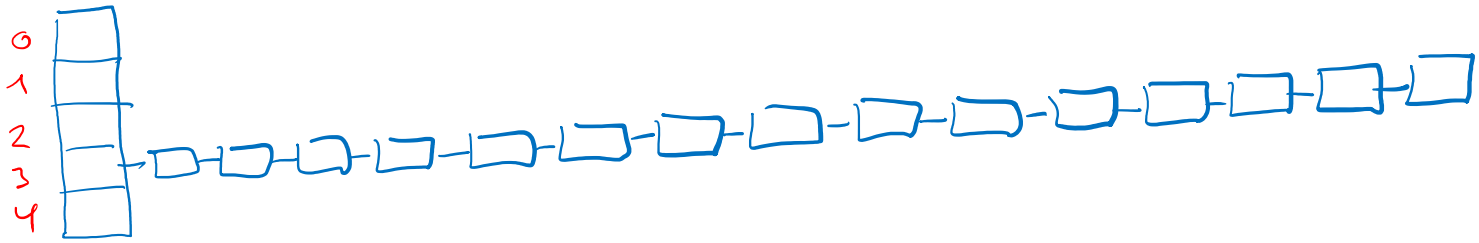
    Falls n = O(m) ist das in Zeit O(1)




  - **Schlechtester Fall ("worst case"):** alle n Schlüssel werden von der Hashfunktion auf denselben Wert abgebildet

    Dann braucht lookup im schlechtesten Fall Θ(n)

    Wie bei der "Realisierung, Idee 1" von Folie 11



# 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 $n / m$)

    Zu den mathematischen Grundlagen dazu gab es bisher eine eigene Vorlesung ("Universelles Hashing") mit einem Crashkurs in Wahrscheinlichkeitsrechnung. Die lassen wir diesmal weg, damit wir mehr Zeit haben für den übrigen Stoff und ein solides Einüben der Grundlagen

    (Sorry an die, die gerne ein höheres Tempo und mehr Stoff hätten, aber man kann nicht alles haben im Leben)

# Hash Map 7/8

- 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 $\underset{1}{21}, \underset{1}{11}, \underset{1}{51}, \underset{1}{71}, \underset{1}{61}, \ldots$

    Zum Umgang mit so einem Fall, siehe Folien 23 – 26

    Für das Ü5 können Sie trotzdem $h(x) = x \bmod m$ nehmen … und feste daran glauben, dass es klappt

    (Sie müssen sich aber für das Ü5 überlegen, wie Sie aus einer Zeichenkette eine Zahl $x$ machen, auf die Sie dann $h(x) = x \bmod m$ anwenden können, mehr dazu auf der nächsten Folie)

# Hash Map 8/8

- Schlüssel, die keine Zahlen sind
  - Sei Objekt in $k$ Bytes $B_0 \ldots B_{k-1}$ repräsentiert

    Bei einem String kann man einfach die ASCII Codes nehmen
  - Dann entspricht der Inhalt dieser Bytes **eindeutig** der Zahl

    $\Sigma_{j=0,\ldots,k-1} B_j \cdot p^j$ eindeutig genau dann wenn $p \geq 256$
  - Diese Zahl ist sehr groß: bei einem integer Datentyp mit $w$ Bytes gibt es Überlauf und man rechnet effektiv mod $256^w$
  - Aber OK, weil wir den Wert dann sowieso mod $m$ hashen

    Probieren Sie für das Ü5 selber aus, was gut funktioniert

    Achtung: die Summe der ASCII-Codes ist keine gute Hashfunktion (klappt nur für kleine Hashtabellen oder sehr lange strings, da sonst keine großen Zahlen herauskommen)

# Rehash 1/4

- Bisherige Annahme
  - Die Schlüsselmenge $S$ ist vorher bekannt
  - Dann kann man 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

    In dem Fall 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

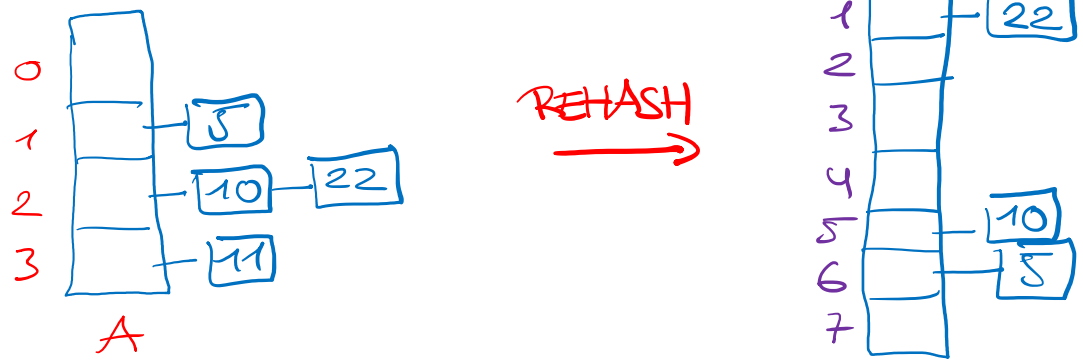
    Das Gute daran: beides kann man leicht feststellen

# Rehash 2/4

- 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 (unter Verwendung der neuen Hashfunktion)
    3. Die alte Tabelle löschen
  - **Beispiel:** $S = \{5, 10, 11, 22\}$, alte Funktion $h(x) = x \bmod 4$, neue Funktion $h(x) = 3 \cdot x - 1 \bmod 8$

$(3 \cdot 11 - 1) \bmod 8 = 32 \bmod 8 = 0$

$(3 \cdot (11 \bmod 8) - 1) \% 8 = (3 \cdot 3 - 1) \% 8 = 0$



# 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 gut gewählten Hashfunktionen ist das unwahrscheinlich

    Wenn die Hashtabelle zu klein geworden ist und man die neue Hashtabelle doppelt so groß wählt (**m → 2m**), dauert es lange, bis man sie wieder vergrößern muss

    Diese "Verdoppelungsstrategie" analysieren wir in der nächsten Vorlesung genauer (amortisierte Analyse)

    Aufgabe 1 vom Ü5 können Sie **ohne** Rehash implementieren

# Rehash 4/4

- Verkleinerung der Schlüsselmenge
  - Die Schlüsselmenge kann auch wieder kleiner werden, indem Schlüssel gelöscht werden
  - Wenn $|S| \ll m$ wird, kann man die Hashtabelle auch wieder verkleinern … siehe ebenfalls nächste Vorlesung
  - Macht man aber in der Praxis oft nicht, weil:

    In sehr vielen Anwendungen braucht man nur **insert** und **lookup**, kein **erase**

    Und selbst wenn man erase braucht, kommt es eher selten vor, dass eine Hash Map zwischenzeitlich sehr viele Schlüssel enthält, man dann die meisten Schlüssel löscht und die Hash Map trotzdem weiterbenutzt

# Ass. Felder in Python, Java, C++ 1/5

- Python ... da heißt ein assoziatives Feld **Dictionary**
  - Grundoperationen — keine Typinformation nötig

| | |
|---|---|
| `map = {}` | Erzeuge leere Map |
| `map[key] = value` | Einfügen von key mit Wert value |
| `value = map[key]` | Wert für key |
| `if key in map: …` | Fragen, ob key enthalten ist |
| `del map[key]` | Löschen von key |
| `list(maps.items())` | Alle key, value Paare als Feld |

Auch oft nützlich ist `map.get(key, default)` ... das ist wie `map[key]`, falls es den `key` gibt, sonst default

# Ass. Felder in Python, Java, C++ 2/5

- Java ... da heißt ein assoziatives Feld **Map**
  - Grundoperationen — K = key type, V = value type

| | |
|---|---|
| `Map<K, V> map = …` | Erzeuge leere Map |
| `map.put(key, value);` | Einfügen von key mit Wert value |
| `value = map.get(key);` | Wert für key |
| `if (map.containsKey(key)) …` | Fragen, ob key enthalten ist |
| `map.remove(key);` | Löschen von key |
| `map.entrySet();` | Alle key, value Paare |

# Ass. Felder in Python, Java, C++ 3/5

- C++ ... da heißt ein assoziatives Feld ebenfalls **Map**
  - Grundoperationen

| | K = key type, V = value type |
|---|---|
| `std::map<K, V> map;` | Erzeuge leere Map |
| `map[key] = value;` | Einfügen von key mit Wert value |
| `value = map[key];` | Wert für key |
| `if (map.contains(key)) …` | Fragen, ob key enthalten ist |
| `map.erase(key)` | Löschen von key |
| `for (auto item : map) …` | Iteration über alle key, value Paare |

# Ass. Felder in Python, Java, C++ 4/5

- C++
  - Achtung bei folgendem Nebeneffekt:

    `if (map[key]) ...` — Fügt key mit Wert 0 ein, falls key bisher nicht in map war

    `if (map.contains(key)) …` — Fragt nur, ob key in map ist

  - Das ist gefährlich, kann aber auch nützlich sein, z.B.

    `map[key]++;` — Erhöht den Wert von key; falls key noch nicht in map, wird vorher mit 0 initialisiert

  - Denselben Effekt kriegt man in Python mit

    ```
    map[key] = map.get(key, 0) + 1
    ```

# Ass. Felder in Python, Java, C++ 5/5

- Effizienz
  - Hängt von der Implementierung ab; effizient sind

    **Hashtabellen** Vorlesung 5

    **Suchbäume** Vorlesung 8
  - In den diversen Programmiersprachen:

    **Java:** java.util.HashMap und java.util.TreeMap

    **C++:** std::unordered_map und std::map

    **Python:** ist dem Compiler überlassen

# Literatur / Links

- Assoziative Arrays
  - In Mehlhorn/Sanders:

    4 Hash Tables and Associative Arrays
  - In Wikipedia

    http://de.wikipedia.org/wiki/Assoziatives_Feld

    http://en.wikipedia.org/wiki/Associative_array
  - In Python, Java, C++

    http://docs.python.org/tutorial

    http://docs.oracle.com/javase

    http://www.cplusplus.com/reference