Technische Hochschule Rosenheim
Technical University of Applied Sciences

Algorithmen und Datenstrukturen
Kapitel 5: Hashtabellen

Prof. Dr. Wolfgang Mühlbauer
Fakultät für Informatik
wolfgang.muehlbauer@th-rosenheim.de

Wintersemester 2019/2020

Eine nicht ernst gemeinte Hashtabelle

Hashtabelle

#

Übersicht

- **ADT Map als Hashtabelle**
  - Tabellen mit indirekter Adressierung, Hashfunktion, Kollision
- Wahl der Hashfunktion
- Kollisionsauflösung
  - durch Verkettung
  - durch Sondieren ("Probing")
- Zusammenfassung und Ausblick

ADT Map

- **Map, Dictionary, Symboltabelle (dt. "assoziatives Datenfeld")**
  - Speichert Key-Value Pairs (dt. "Schlüssel-Werte-Paare")
  - Bsp.: Alter von Personen → { (Trump, 73), (Merkel, 65), (Kurz, 33) }

- **Typische Operationen**
  - `void put(Key key, Value value)` (oft auch "`insert`")
  - `Value get(Key key)`
  - `void delete(Key key)` (oft auch "`remove`")
  - `boolean contains (Key key)`
  - `boolean IsEmpty()`
  - `int size`
  - `Iterable<Key> keys` (durchlaufe alle Schlüssel)

- **Annahmen**
  - Jeder Wert hat einen Key
  - Keys sind eindeutig, keine Duplikate!
  - Trägt man den selben Schlüssel nochmals ein, wird der bisherige Wert überschrieben.
  - Weder der Key noch der Value darf `null` sein.

Anwendungen von Maps

Anwendung	Zweck der Suche	Schlüssel	Wert
Wörterbuch	Finde eine Definition	Wort	Definition
Index in einem Buch	Finde Seiten, die Suchbegriff enthalten	Suchbegriff	Liste mit Seitenzahlen
Websuche	Suche relevante Webseiten	Schlüsselwort	Liste der URLs
Compiler	Finde Typ und Wert	Variablenname	Typ, Wert, Adresse

```
i       int       0x87C50FA4
j       int       0x87C50FA8
x       double    0x87C50FAC
name    String    0x87C50FB2
```

Compiler

```
EDITOR=emacs
GROUP=mitarbeiter
HOST=vulcano
HOSTTYPE=sun4
LPDEST=hp5
MACHTYPE=sparc
```

Umgebungsvariablen

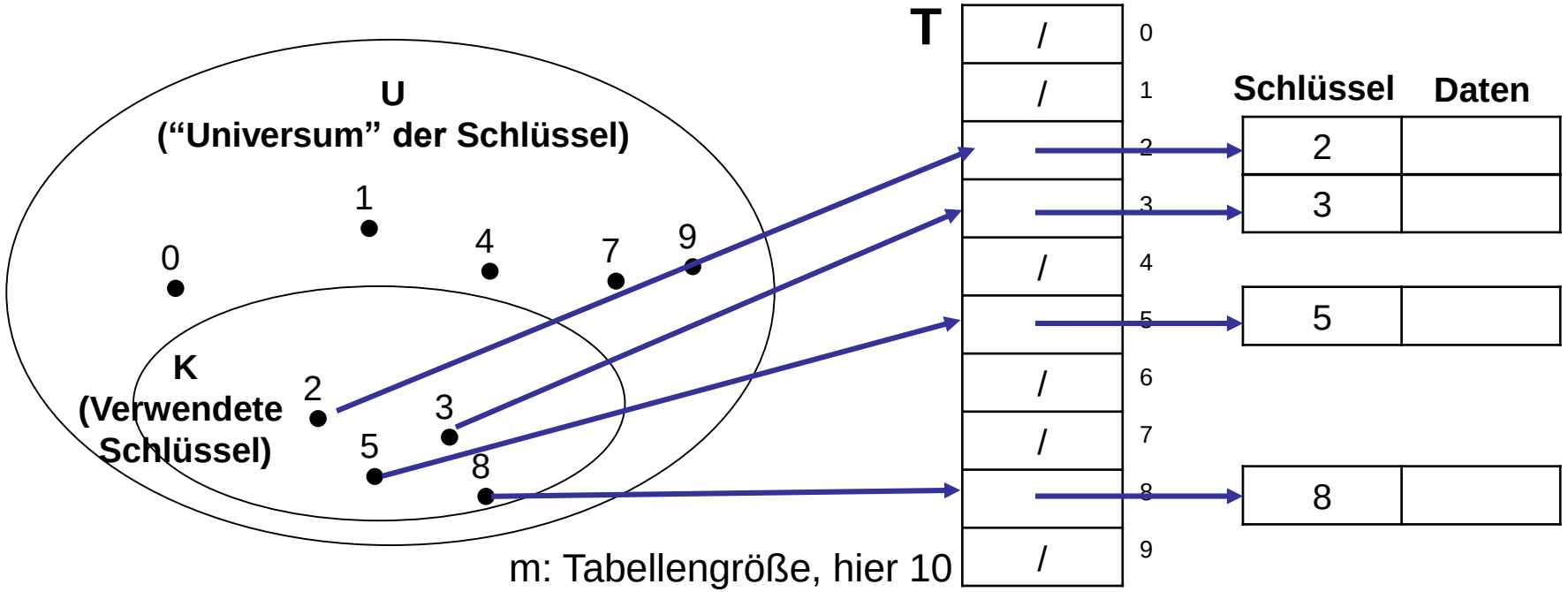
Hashtabellen

- Hashtabellen erlauben eine effiziente Implementierung der ADT Map.
  - Durchschnittliche Laufzeit der Suche: $\boldsymbol{O(1)}$
  - Worst Case Laufzeit der Suche: $\boldsymbol{\Theta(n)}$ → tritt selten ein

- **Idee:** Tabellen mit indirekter Adressierung
  - Berechne Ort an dem ein Datensatz gespeichert ist.
  - Ähnlich wie bei einem Array.

Tabellen mit direkter Adressierung (1)

- **Szenario / Annahmen**
  - *Dynamische* Elementmenge
  - Jedes Element hat Schlüssel aus Universum $U = \{0,1,\ldots,\boldsymbol{m}-1\}$
  - 2 Elemente haben nie den gleichen Schlüssel!
- Repräsentiere Menge als ***Adresstabelle / Array*** $T[0\ldots m-1]$
  - Jede Position entspricht Schlüssel aus *U*.
  - $T[k]$ enthält Zeiger auf *x*, falls Element *x* mit Schlüssel *k* vorhanden ist; ansonsten ist $T[k]$ leer.



Tabellen mit direkter Adressierung (2)

- Einfache Implementierung der ADT "Menge".

- **Nachteile:**
  - Was passiert, falls *U* sehr groß ist?
  - Was passiert, falls Tabellengröße ***m*** viel kleiner als $|U|$?

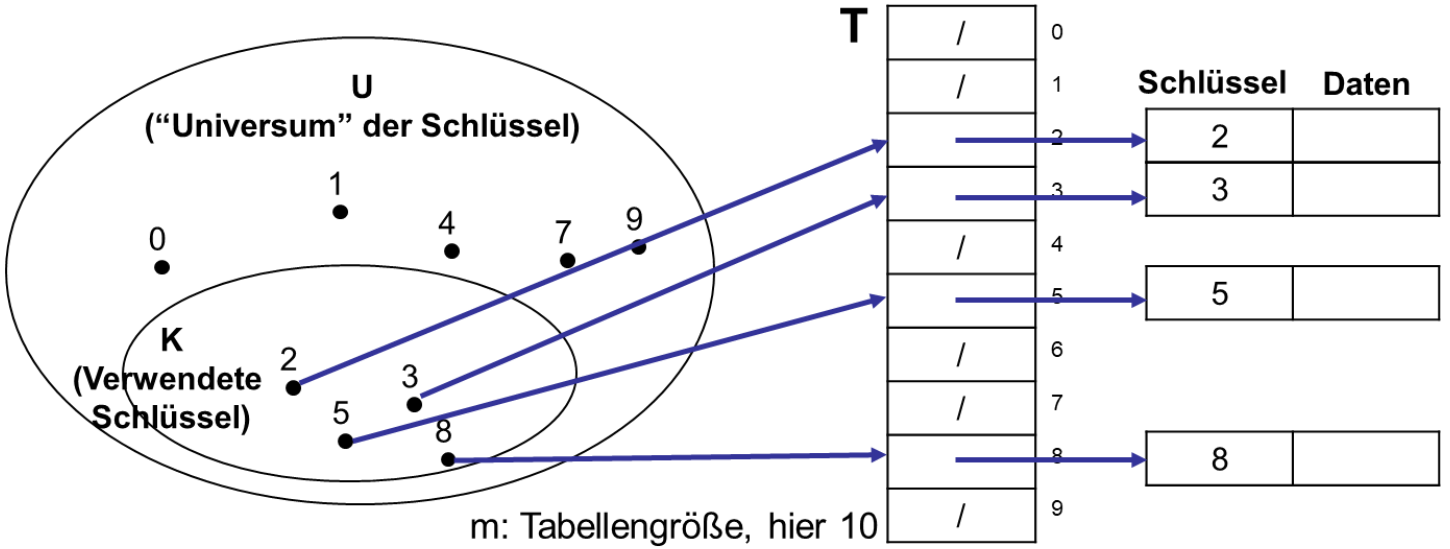
DIRECT-ADDRESS-TABLES

```
GET(key)
    return T[key]
```

value

```
PUT(key, value)
    T[key] = value

DELETE(key)
    T[key] = null
```

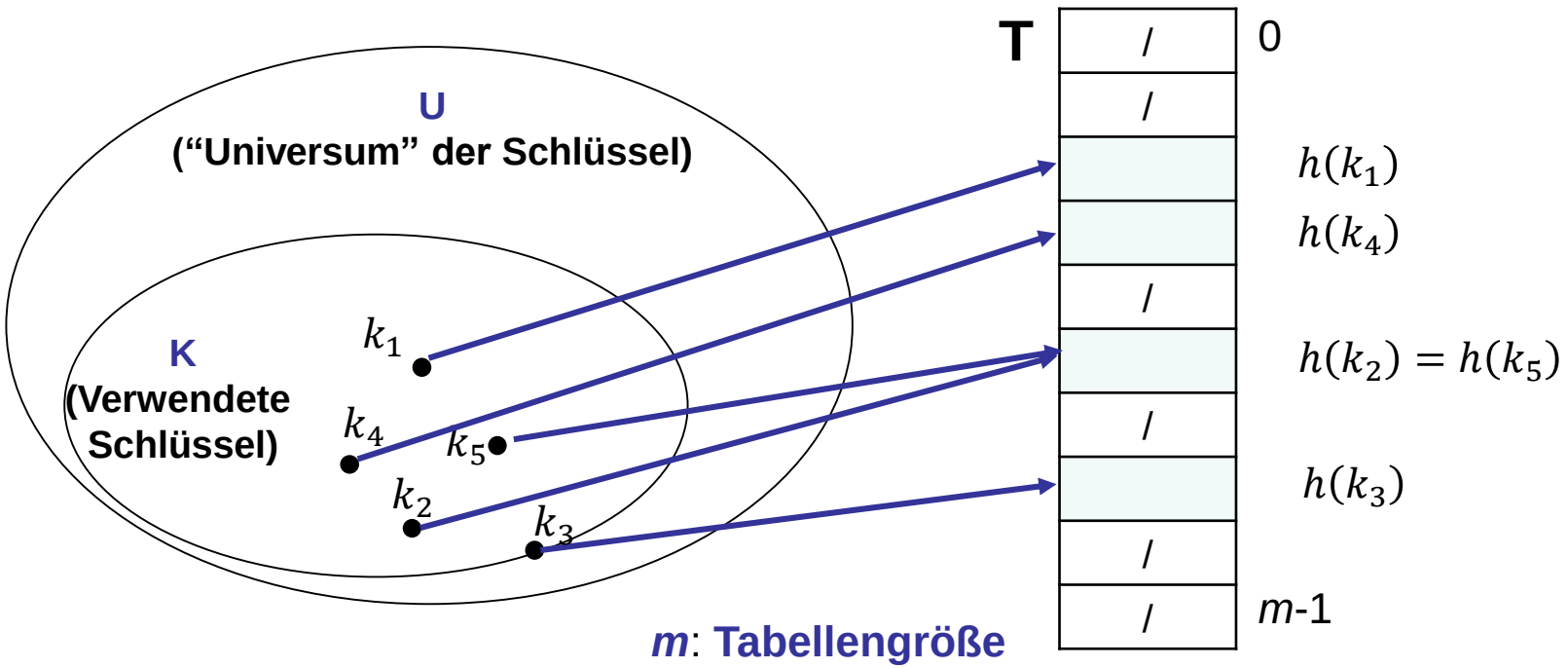


Hashtabellen / Indirekte Adressierung

- Hashtabellen: Tabellengröße m **viel kleiner als $|U|$!**
  - Speicherbedarf: $\Theta(|m|)$
  - Laufzeit: $O(1)$ im Durchschnitt

Idee

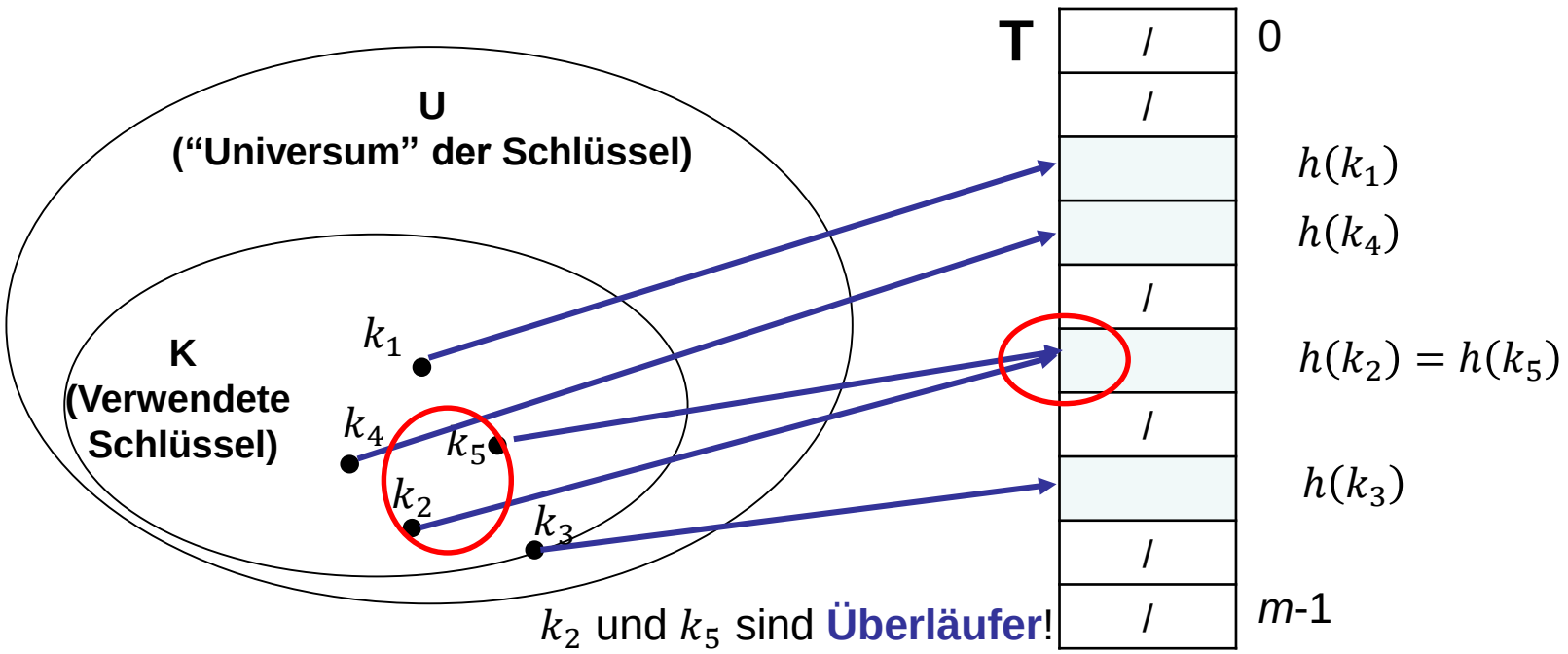
- Verwende ***Hashfunktion*** *h* und speichere Element an Position *h(k)*
- $h : U \to \{0, 1, \ldots, m-1\}$, so dass *h(k)* ein vorhandener Slot in *T* ist.
- *h* „hasht“ Schlüssel *k* auf Slot *h(k)*



Hashtabellen: Kollisionen

- **Problem „Kollision“:** ≥ 2 Schlüssel fallen auf die gleiche Arrayposition
  - $|U| > m$, mehr mögliche Schlüssel als Arraygröße: Kollisionen können auftreten
  - $|K| > m,$ mehr Schlüssel als Arraygröße: Kollisionen müssen auftreten.

- Kollisionsauflösung: Kollisionen müssen abgefangen werden.
  - Strategien: ***Verkettung*** und ***Sondieren.***



Herausforderungen beim Hashing

- **Wahl einer „guten“ Hashfunktion**
  - Vermeidung von Kollisionen
  - Hashfunktion soll auftretende Schlüssel möglichst gleich verteilen.
  - Dennoch: Kollisionen sind in der Praxis unvermeidbar und müssen abgefangen werden.

- **Wie geht man mit Kollisionen um?**
  - Verkettung der Überläufer
  - Sondieren / Probing

- **Wie wählt man Größe der Hashtabelle?**
  - Wählt man *m* zu groß, gibt es viel ungenutzten Speicherplatz.
  - Wählt man *m* zu klein, gibt es viele Kollisionen.
  - **Belegungsfaktor** $\alpha = \frac{\#\text{ gespeicherte Schlüssel}}{\text{Größe der Hashtabelle}} = \frac{|K|}{m} = \frac{n}{m}$

Publikums-Joker: Hashtabellen

Welche der folgenden Aussagen ist ***falsch***?

A. Die ADT Map lässt sich auch durch eine verkettete Liste implementieren.

B. Hashfunktionen müssen effizient berechenbar sein.

C. Ähnliche große Schlüssel liegen bei Hashtabellen hintereinander im Speicher.

D. Mit Hashtabellen kann mit die ADT Map UND die ADT Set implementieren.

Übersicht

- ADT Map als Hashtabelle
  - Tabellen mit indirekter Adressierung, Hashfunktion, Kollision
- **Wahl der Hashfunktion**
- Kollisionsauflösung
  - durch Verkettung
  - durch Sondieren ("Probing")
- Zusammenfassung und Ausblick

Wahl der Hashfunktion

- **Anforderungen** an eine Hashfunktion
  - Leicht und schnell berechenbar.
  - Verteilt Schlüssel **gleichmäßig** über die Tabelle.
    - Problem: Häufigkeitsverteilung der Schlüssel vorab meist nicht bekannt.
  - Das Ergebnis sollte für ähnliche Schlüssel unterschiedlich sein.
    - In der Praxis kommen nämlich ähnliche Schlüssel häufig vor.

- ***Mögliche Hashfunktionen***
  - Divisionsmethode
  - Multiplikationsmethode

- **Wichtig:** Kollisionen dennoch unvermeidbar!

- **Annahme:** Schlüssel sind positive, ganze Zahlen.
  - Was wenn nicht? → siehe nächste Folie

Hashfunktionen in der Praxis

- ***Hashfunktion h:*** $\mathbb{Z}^+ \rightarrow [0..m]$
  - Berechnet für positive ganze Zahlen einen Arrayindex (= Position in der Hashtabelle)

- ***hashCode***: $Java\ Object \rightarrow \mathbb{Z}^+$
  - Wenn Schlüssel keine positive ganze Zahl, so muss dieser erst in eine solche umgewandelt werden
  - In Java muss dazu für die Schlüsselklasse **`hashCode()`** implementiert werden.
    - `hashCode` der Klasse Student liefert die Matrikelnummer
    - `hashCode` für Double: XOR der vorderen 32 Bits mit den hinteren 32 Bits der Bitrepräsentation.

- ***Zusammengesetzte Datentypen***
  - Mische die einzelnen Felder zusammen.
  - *String:* Mische Zeichen
  - *Datum:* Mische Tag, Monat, Jahr
  - Richtlinie: Wähle für *R* eine Primzahl, die klein genug ist, um Overflows zu vermeiden, z.B. 31.

String

```
int hash = 0;
for (int i = 0; i < s.length(); i++)
    hash = (R * hash + s.charAt(i)) % m
```

Datum

```
int hash = (((day * R + month) % m) * R
+ year) % m
```

Divisionsmethode

Hashfunktion: $h(k) = k \bmod m$

- **Vorteil**
  - Schnell, benötigt nur 1 Division.
- **Nachteil**
  - Manche Werte von *m* sollten vermieden werden.
  - 2er Potenzen sind schlecht:
    - Falls $m = 2^p$, dann entspricht das Ergebnis von *h(k)* den *p* Least Significant Bits von *k*.
    - Ähnliche große Zahlen werden dann auf den gleichen Wert gehasht.
- Gute Wahl von *m:* Primzahl

Ausblick: Multiplikationsmethode

Hashfunktion: $h(k) = \lfloor m \cdot (k \cdot A \mod 1) \rfloor$

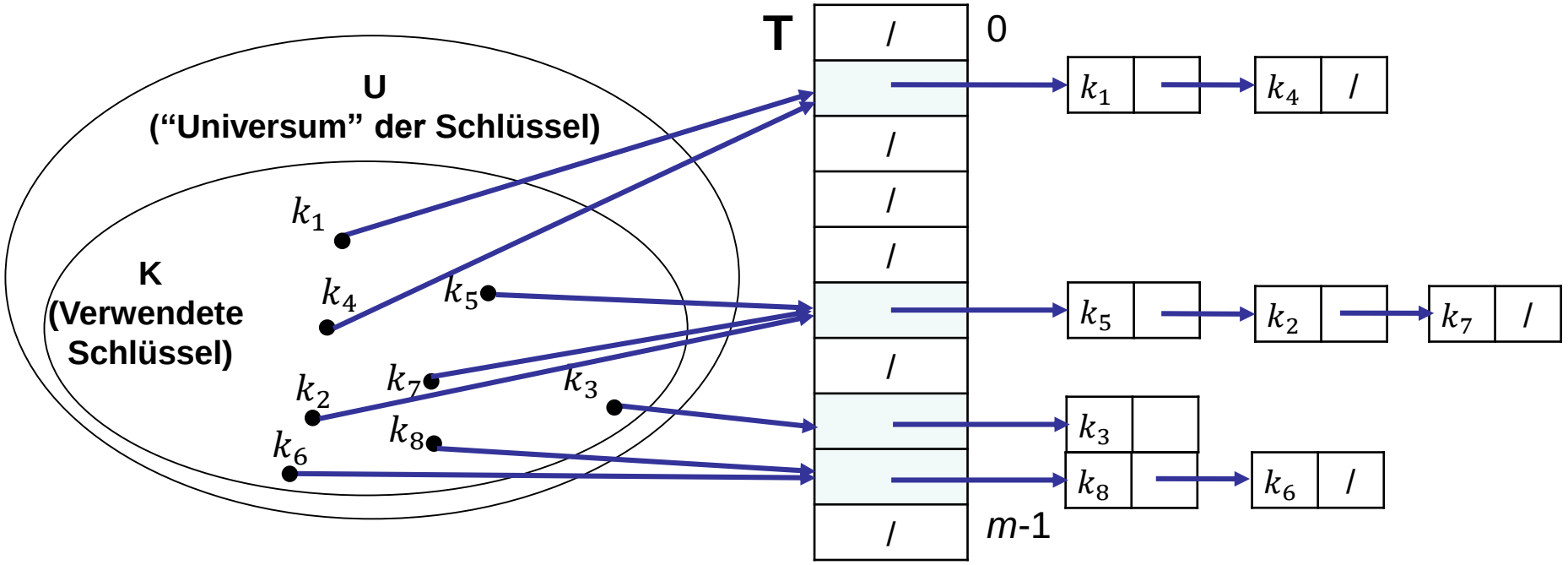
- **Berechnung**
  - Multipliziere Schlüssel *k* mit selbst gewählter Konstante $0 < A < 1$.
  - Multipliziere Bruchanteil des Ergebnisses mit Größe der Hashtabelle *m*.
  - Runde Ergebnis ab auf ganze Zahl.
- **Vorteil:** Der Wahl von *m* ist nicht kritisch.
- **Nachteil**: Mehr Rechenaufwand als bei Divisionsmethode.
- Hinweise:
  - Nur $h(k) = \lfloor m \cdot (k \mod 1) \rfloor$ (ohne *A*) würde mehr Gewicht auf die höherwertigen Bits legen.

Übersicht

- ADT Map als Hashtabelle
  - Tabellen mit indirekter Adressierung, Hashfunktion, Kollision
- Wahl der Hashfunktion
- **Kollisionsauflösung**
  - **durch Verkettung**
  - durch Sondieren ("Probing")
- Zusammenfassung und Ausblick

Kollisionsauflösung durch Verkettung ("Chaining")

- **Idee**
  - Alle Elemente, die auf gleiche Position „gehasht“ werden (=***Synonyme***), werden in einer verketteter Liste.
  - Position *i* der Tabelle enthält Zeiger auf Anfang der verketteten Liste.
  - Falls es keine solchen Elemente gibt, Zeiger auf NIL bzw. null.
- **Animation:**
  https://www.cs.usfca.edu/~galles/visualization/OpenHash.html



Implementierung und erste Laufzeitanalyse

```
HASHTABLES-CHAINING
let heads[0..(m-1)] be array that points to first nodes of linked lists
PUT(key, val)
    i = key.hashCode() & 0x7FFFFFFF) % m
    add (key,val) to end of linked list stored at heads[i]
GET(key)
    i = key.hashCode() & 0x7FFFFFFF) % m
    find key in linked list stored at heads[i] and return val
DELETE(key)
    i = key.hashCode() & 0x7FFFFFFF) % m
    delete (key,val) from linked list stored at heads[i]
```

Quellcode: HashTableChaining.java

- **Rehashing**
  - Schätze durchschnittliche Listenlänge ab → n / m
  - Falls Listen sehr lang, kopiert `put` alle Elemente in eine neue größere Hashtabelle.
- **Einfügen**
  - Ist Schlüssel bereits enthalten, dann überschreibe vorhandenen Wert.
  - Fall `val == null`, dann lösche den dazugehörigen Schlüssel.
  - Ggfs. Hashtabelle vergrößern / Rehashing
- **Löschen**
  - Ggfs. Hashtabelle vergrößern / Rehashing

Laufzeit

- Laufzeit abhängig vom **Belegungsfaktor** $\boldsymbol{\alpha = \frac{n}{m}}$
  - Entspricht durchschnittlicher Listenlänge
  - *n*: Anzahl der Elemente in Tabelle
  - *m*: Anzahl der Tabellenplätze = Anzahl der verketteten Listen
  - Es kann gelten: $\alpha < 1,\ \alpha = 1,\ \alpha > 1$

- **Worst** Case für **ein einzelne `get`-Operation: O(n)**
  - alle Elemente der Hashtabelle sind in der gleichen Liste UND
  - das gesuchte Element steht am Ende der Liste oder ist gar nicht gespeichert.

- **Best** Case für **ein einzelne `get`-Operation: O(1)**
  - die benötigte Liste enthält genau 1 bzw. kein Element ODER
  - der gesuchte Schlüssel steht ganz am Anfang der Liste.

- **Durchschnittliche Kosten bei einer Folge von t Operationen**: $\boldsymbol{O(t \cdot n/m)}$
  - "Amortisierte Kosten" falls jeder Schlüssel mit gleicher W'keit gesucht wird.

- **Rehashing / Array Resizing**
  - Rehashing versucht $\frac{n}{m}$ klein zu halten.
  - Damit wird die erwartete/amortisierte Laufzeit für eine einzelne Operation $\boldsymbol{O(1)}$

Publikums-Joker:

Welche der folgenden Aussagen ist ***falsch***?

A. Belegungsfaktoren von $\alpha = n/m > 1$ stellen kein Problem dar, falls Kollisionsauflösung durch Verkettung eingesetzt wird.

B. Der Vorteil von Hashtabellen ist, dass sie stets eine Suche in O(1) erlauben.

C. Es ist teuer, alle Schlüssel einer Hashtabelle in aufsteigender Reihenfolge zu durchlaufen (Annahme: Es gibt eine Ordnung für die Schlüssel)

D. Werden Hashtabelle nicht dynamisch vergrößert ("Rehashing"), dann ist die erwartete Laufzeit in der Regel O(n).

Übersicht

- ADT Map als Hashtabelle
  - Tabellen mit indirekter Adressierung, Hashfunktion, Kollision
- Wahl der Hashfunktion
- **Kollisionsauflösung**
  - durch Verkettung
  - **durch Sondieren ("Probing")**
- Zusammenfassung und Ausblick

Kollisionsauflösung durch Sondieren

- **Idee:** Speichere ***alle*** Schlüssel ***direkt*** in der Tabelle,
  - Überläufer werden an freien Tabellenpositionen gespeichert, nicht in Listen!

- **Problem:** Wie findet man dann später die Überläufer?

- Häufigster Ansatz: **Lineares Sondieren** zum Finden eines Schlüssels *k*
  - "Falls Platz bereits belegt, verwende nächsten Tabelleneintrag"
  - Berechne *h(k),* ***sondiere*** die Tabellenposition *h(k)* → 3 Fälle
    a) Position *e*nthält gesuchten Schlüssel *k* → Suche erfolgreich + Abbruch!
    b) Position *ist leer* → Suche erfolglos + Abbruch!
    c) Position enthält Schlüssel ungleich *k* → Sondiere an ***nächster Position***
  - Falls Fall c) auch bei nächster Sondierung auftritt, wiederhole so lange bis entweder der Schlüssel oder eine leere Tabellenposition gefunden wurde.

- Animation
  - https://www.cs.usfca.edu/~galles/visualization/ClosedHash.html

Lineares Sondieren: Einfügen

- **Beispiel:** m = 7, füge der Reihe nach ein: 12, 53, 5, 15, 2, 19

19	15	2		53	12	5

- Schlüssel und Werte werden in Array gespeichert
  - Keys[i] und vals[i] bilden ein Schlüssel-Werte-Paar

```
// m defines table size, n number of entries in hashtable
Key[] keys = new Keys[m]
Value[] vals = new Value[m]

PUT(key, val)
1  if n ≥ m/2
2     resize(2*m)      // rehashing if > half of table occupied
3  for (i = hash(key); keys[i] != null; i = (i + 1) % m)      // sondiere
4        if keys[i].equals(key)          // key already exists
5           vals[i] = val
6           return
7    keys[i] = key     // found empty position, key doesn't exist yet
```

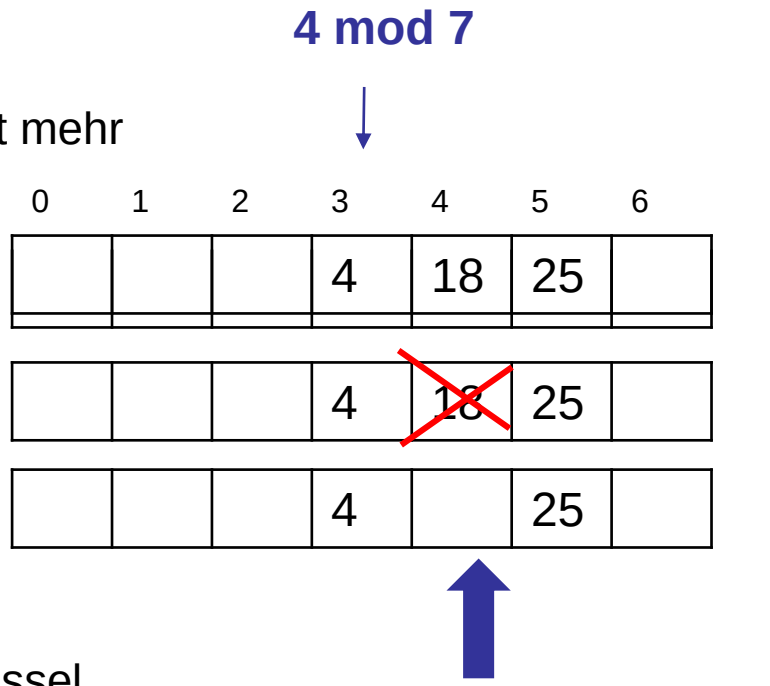
Quellcode: HashTableProbing.java

Hinweis: Get(key) ist analog zu implementieren

Lineares Sondieren: Löschen

- **Vorgehen**
  - Suche zunächst den zu löschenden Eintrag `i`
  - Setze dann `keys[i]` auf `null`

- **Problem**
  - Schlüssel, die danach eingefügt wurden, sind nicht mehr auffindbar.
  - Beispiel:
    - *m* = 7, der Reihe nach eingefügt: 4, 18, 25
    - Für alle Schüssel gilt h(k) = k mod 7 = 4
    - Dann: Löschen von 18
    - Dann: Suche von 25



- **Lösung**
  - Alle Schlüssel zwischen dem zu löschenden Schlüssel und der nächsten freien Position (= Cluster) müssen gelöscht und erneut eingefügt werden.
  - Hier: Schlüssel 25

Lineares Sondieren: Löschen

```
int i = hash(key);
while (!key.equals(keys[i])) {
    i = (i + 1) % m;
}

keys[i] = null;
vals[i] = null;

i = (i + 1) % m;
while (keys[i] != null) {   // Clusterende
    Key   keyToRehash = keys[i];
    Value valToRehash = vals[i];
    keys[i] = null;
    vals[i] = null;
    n--;
    put(keyToRehash, valToRehash);
    i = (i + 1) % m;
}
 n--;
```

Suche zu löschenden Schlüssel

Lösche Schlüssel und Wert

Lösche alle Schüssel danach (bis man eine leere Position findet, "Clusterende") und trage diese erneut in Hashtabelle ein

Diskussion: Lineares Sondieren

- ***Clustering***
  - Große Cluster wachsen schneller als kleine.
  - Beispiel: Bei Einfügen ist die Wahrscheinlichkeit am höchsten, dass Position / Index 4 belegt wird. (falls Schlüssel gleich wahrscheinlich)

0	1	2	3	4	5	6	7	8	9
	1	12	3					8	

mod 10

- ***Array resizing***
  - `put` (`delete`) vergrößert (verkleinert) dynamisch die Hashtabelle.
  - Man muss darauf achten, dass Tabelle höchstens zur Hälfte belegt ist.

- ***Amortisierte Laufzeit***
  - Erwartete Kosten für Folge von ***t*** Operationen falls Array Resizing dafür sorgt, dass Tabelle höchstens zur Hälfte belegt ist: **O(t)** (ohne Beweis)
  - Im Mittel annähernd **O(1)** pro `put, get, delete`

Übersicht

- ADT Map als Hashtabelle
  - Tabellen mit indirekter Adressierung, Hashfunktion, Kollision
- Wahl der Hashfunktion
- Kollisionsauflösung
  - durch Verkettung
  - Durch Sondieren ("Probing")
- **Zusammenfassung und Ausblick**

Verkettung vs. Probing

- **Vorteile: Verkettung**
  - Belegungsfaktor von $\alpha > 1$ möglich, also mehr *n* größer als Tabellengröße *m*
  - Einfacher zu implementieren.
  - Vergrößern der Hashtabelle ("Rehashing") bei vielen Elementen nicht zwingend nötig.

- **Vorteile: Probing**
  - Kein Overhead durch Verkettung
  - Kein ***new***-Operator / Anfordern von Speicher bei Einfügen.
  - Cache-Lokalität: Alle Daten liegen hintereinander im Speicher (Arrays!)

- In der Praxis (auch Java) wird meist Verkettung verwendet, obwohl Probing theoretisch performanter ist.
  - https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/HashMap.html

- Wichtige Parameter für Programmierer:
  - **`capacity`**: Initiale Größe der Hashtabelle
  - **`load factor`**: Belegungsfaktor, bestimmt wann Tabelle vergrößert bzw. verkleinert werden muss.

Publikums-Joker:

Welche der folgenden Aussagen ist ***falsch***?

A. Hashfunktionen müssen deterministisch sein.

B. In welcher Reihenfolge die Schlüssel in der Hashtabelle stehen ist bei linearem Sondieren davon abhängig, in welcher Reihenfolge sie eingefügt wurden.

C. Beim linearen Sondieren ist Rehashing / Array Resizing unnötig.

D. Löschen ist bei Verkettung der Überläufer einfacher als bei linearem Sondieren.

Terminologie

- Verkettung
  - Heißt manchmal „Open Hashing“
  - https://www.cs.usfca.edu/~galles/visualization/OpenHash.html
- Sondieren / Probing
  - Heißt manchmal „Closed Hashing“
  - https://www.cs.usfca.edu/~galles/visualization/ClosedHash.html

Übersicht

- Hashtabellen
  - Tabelle mit direkter Adressierung
  - Hashing
- Kollisionsauflösung
  - Verkettung
  - Lineares Sondieren
- Wahl der Hashfunktion
  - Divisionsmethode
  - Multiplikative Methode
- Hashing ungeeignet falls
  - man schnell und oft den minimalen/maximalen Schlüssel bestimmen will.
  - man alle Schüssel in einem gewissen Bereich suche möchte
  - man geordnet über alle Elemente laufen möchte.
- Ansonsten ist Hashing sehr performant und sehr weit verbreitet

Quellenverzeichnis

[1] Cormen, Leiserson, Rivest and Stein. *Introduction to Algorithms*, Third Edition, The MIT Press, 2009.

[2] Ottmann, Widmayer. *Algorithmen und Datenstrukturen*, Kapitel 1.2.3, 5. Auflage, Spektrum Akademischer Verlag, 2012. (xxx)

[3] BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=377194 (abgerufen am 11.11.2016)

[4] https://commons.wikimedia.org/wiki/File:Hackfleisch-1.jpg, (abgerufen am 11.11.2016)