Algorithmen und Datenstrukturen

Assoziation von Objekten, Wörterbücher, Hashing
Teil a

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Felix Kuhr (Übungen)

sowie viele Tutoren

Danksagung

Einige der nachfolgenden Präsentationen wurden mit ausdrücklicher Erlaubnis des Autors und mit umfangreichen Änderungen und Ergänzungen übernommen aus:

- „Effiziente Algorithmen und Datenstrukturen“ (Kapitel 4: Hashing) gehalten von Christian Scheideler an der TUM
  http://www14.in.tum.de/lehre/2008WS/ea/index.html.de

- „Algorithmen und Datenstrukturen“
  gehalten von Sven Groppe an der UzL

Wörterbucher / Dictionaries

Mengen von Schlüssel-Wert-Paaren (key, entry)

Spezifikation der Operationen:

- insert(k, e, d): d := d ∪ {(k, e)}
  // Änderung nach außen sichtbar
- delete(k, d): d := d \ {(k, e)}, wobei e das Element ist, das unter dem Schlüssel k eingetragen ist
  // Änderung von d nach außen sichtbar
- lookup(k, d): Falls es ein (k, e) ∈ d gibt, dann gib e aus, sonst gib ⊥ aus

Abstrakter Datentyp Dictionary

- dict := <(1, “eins”), (5, “fünf”), (42, “zweiundvierzig”)>:Dictionary

- Warum verwende ich eine sog. Sequenz <…>
  statt einer Menge {…} oder einer Liste […]?
- Mengen und Listen können verändert werden und ein Ausdruck {…}
  oder […] erzeugt immer eine neue Instanz, wenn er ausgewertet wird
- Ich möchte z.B. in einer Schleife nicht immer neue Objekte erzeugen
- Mit <…> notierte Sequenzen sind nicht veränderbar
- Wir können über Sequenzen iterieren:

  seq := <(1, “eins”), (5, “fünf”), (42, “zweiundvierzig”)>

  for (key, value) ∈ seq **do** …

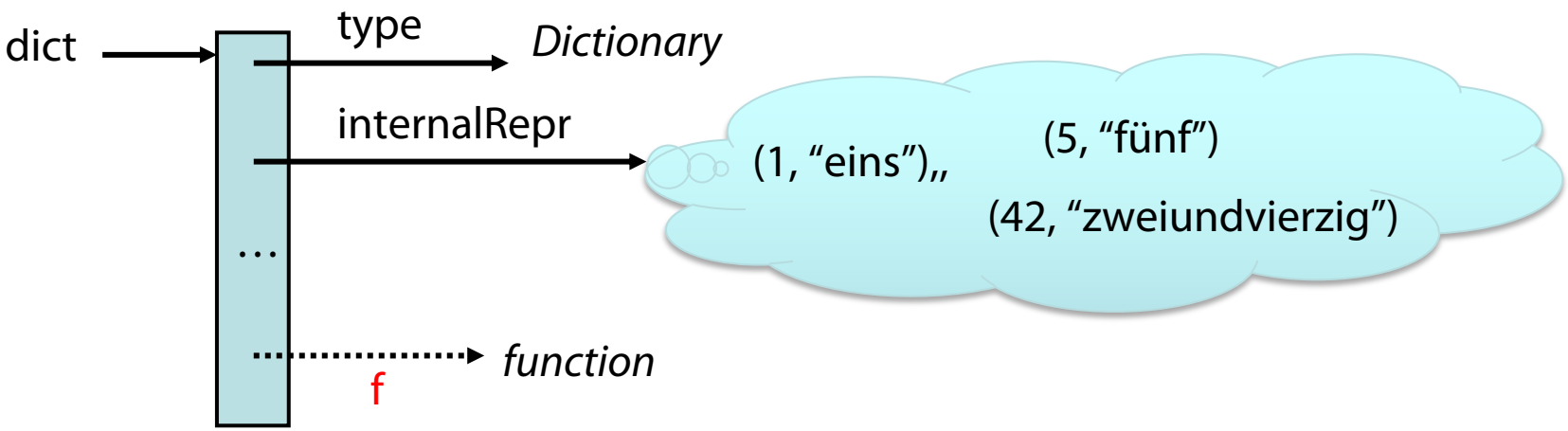
- Wie programmiert man die
  Initialisierung einer neuen Instanz
  eines abstrakten Datentyps?

Wie schon bei der Zuweisung
kann auch bei **for** ein
Musterausdruck stehen

Initialisierung von ADT-Instanzen

type Dictionary () (internalRepr, …)

dict := <(1, “eins”), (5, “fünf”), (42, “zweiundvierzig”)>:Dictionary with f as …



procedure initializeInstance(d:Dictionary, initValues:Sequence)

for (key, value) ∈ initValues **do** …

…

internalRepr(d) := …

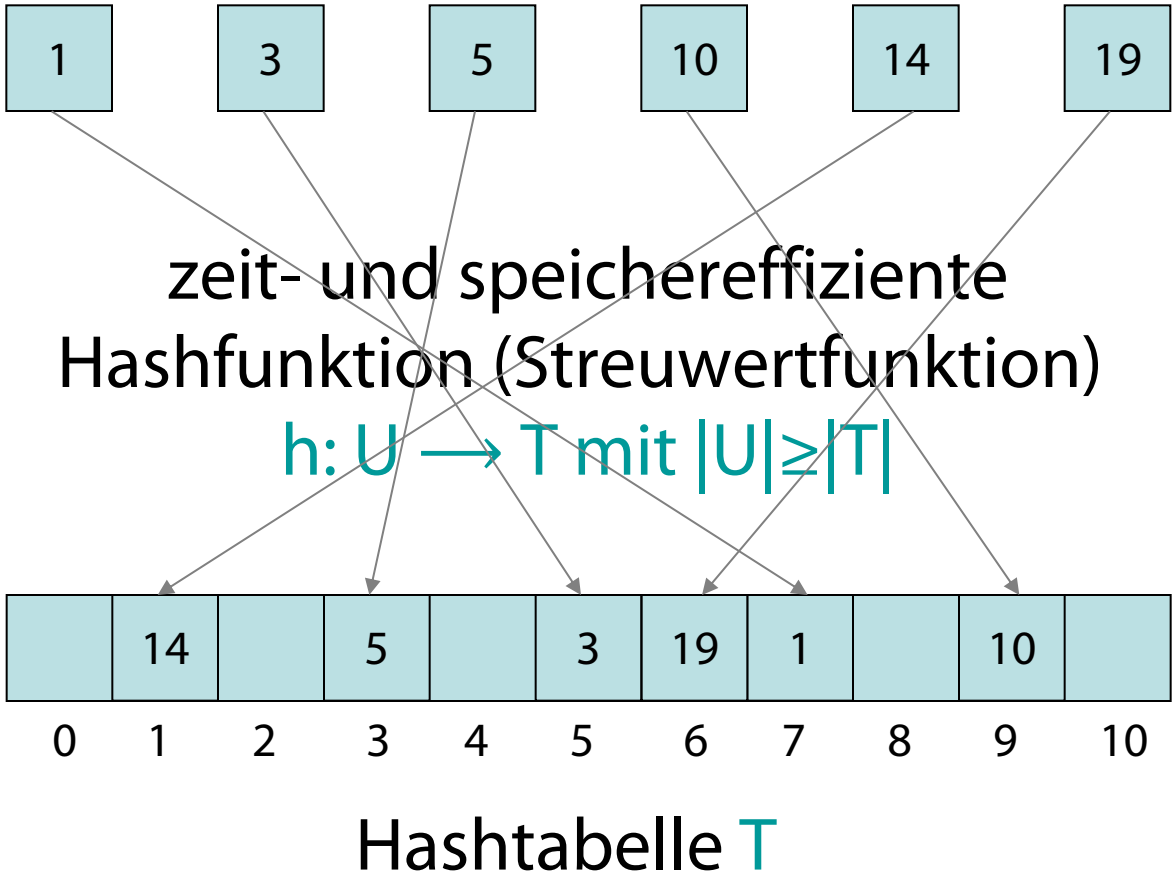
Wörterbücher / Dictionaries

Vergleich zur Menge:

- Als Elemente (Einträge) bei Wörterbüchern nur Attribut-Wert-Paare vorgesehen
- Iteration über Elemente eines Wörterbuchs in **willkürlicher** Reihenfolge **ohne** Angabe eines Bereichs
  - Über alle Attribute (Schlüssel)
    - **for** (key, _) ∈ dict **do** …
  - Über alle Werte
    - **for** (_, value) ∈ dict **do** …
  - Über alle Attribut-Wert-Paare
    - **for** (key, value) ∈ dict **do** …

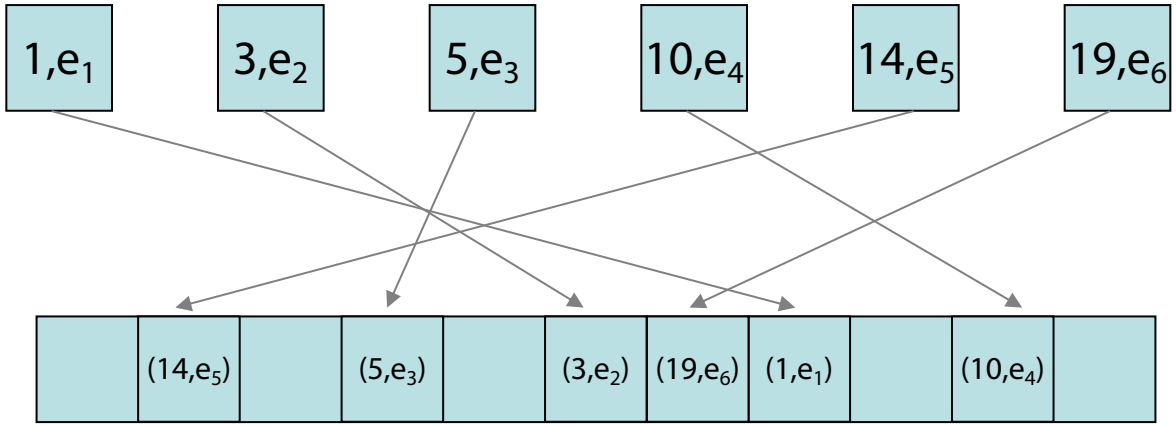
Hashing (Streuung)

Einige Elemente
aus einer Menge U:



Assoziation durch Hashing

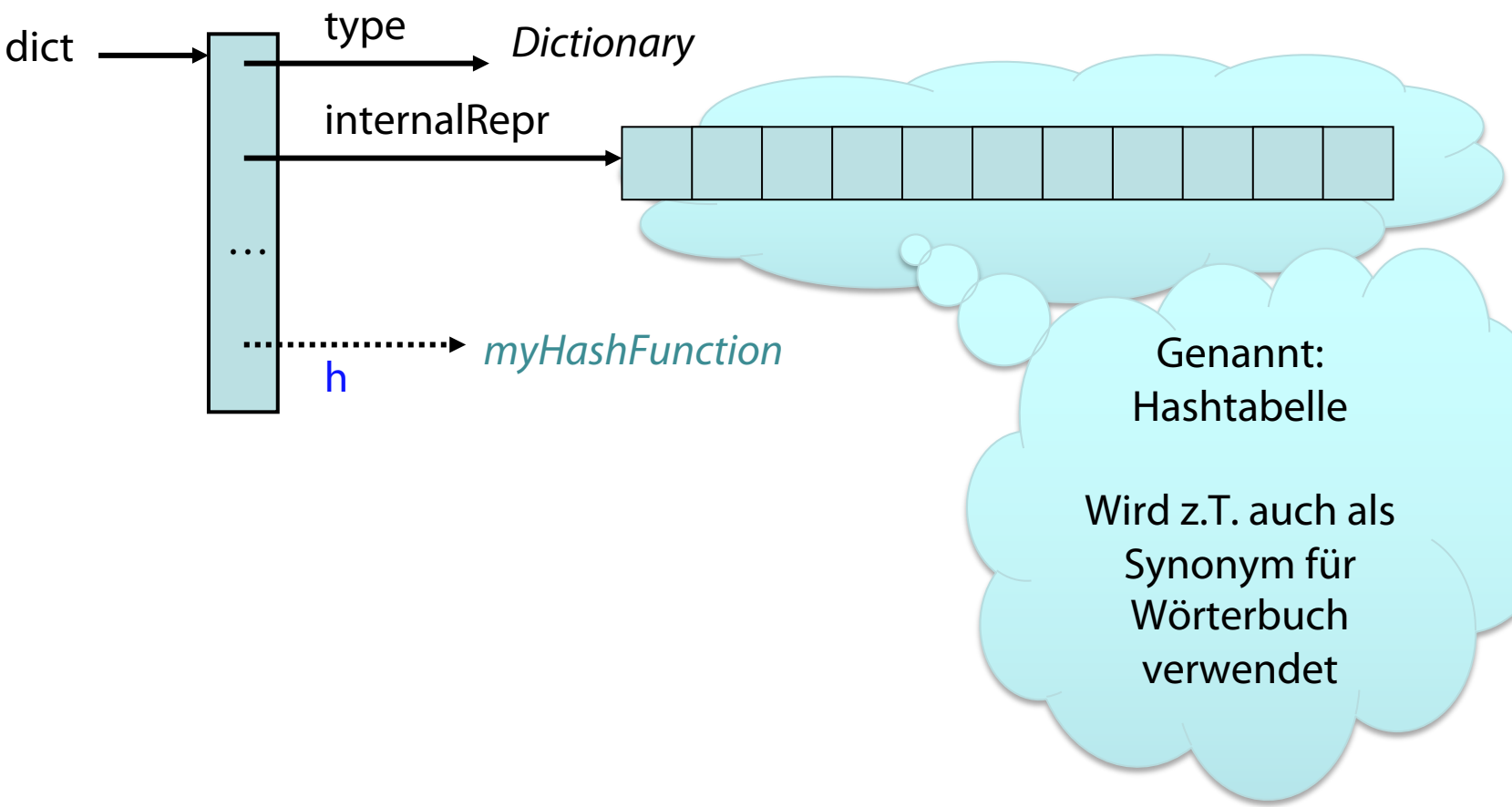
- Schlüssel k seien hier Zahlen aus einem großen Bereich
- Assoziation von k mit e



- Schlüssel k selbst können auch Objekte sein, es muss nur eine Abbildung auf T definiert sein bzw. werden (Dictionary-spezifische Hash-Funktionen)

Initialisierung von ADT-Instanzen

dict := <…>:Dictionary with h as myHashFunction

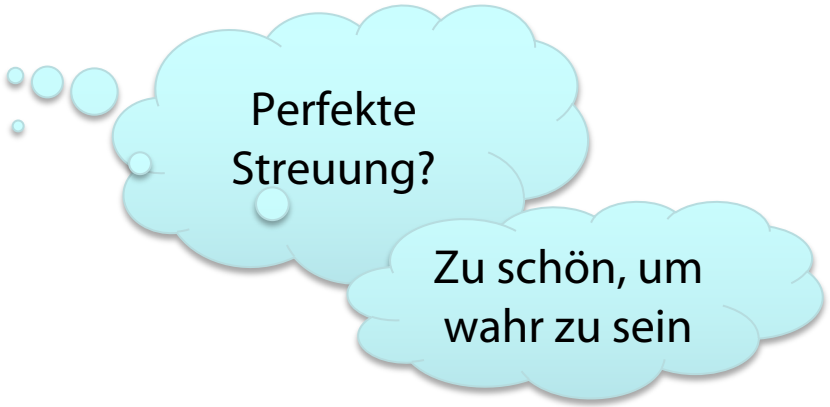


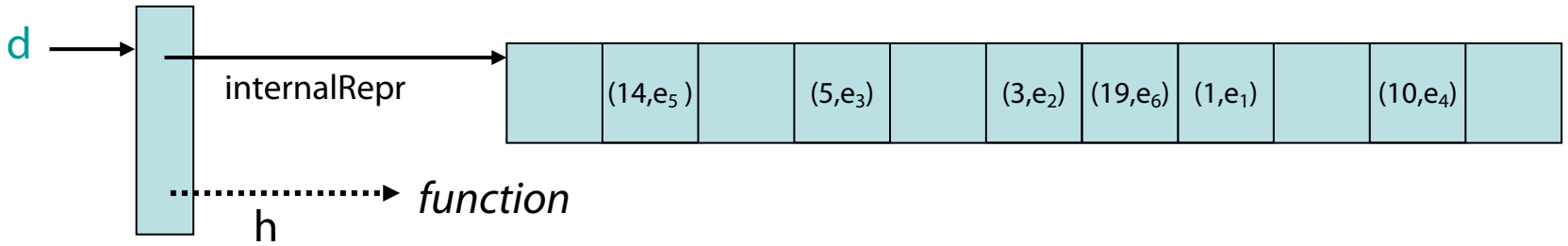
Hashing (perfekte Streuung, kein Kollisionen)

procedure insert(k, e, d:Dictionary)
    T := internalRepr(d)
    T[h(k)] := (k,e)

procedure delete(k, d:Dictionary)
    T := internalRepr(d)
    T[h(k)] := ⊥

function lookup(k, d:Dictionary)
    T := internalRepr(d); t := T[h(k)]
    if t = ⊥ **then return** ⊥ **else return** second(t)





Einschub

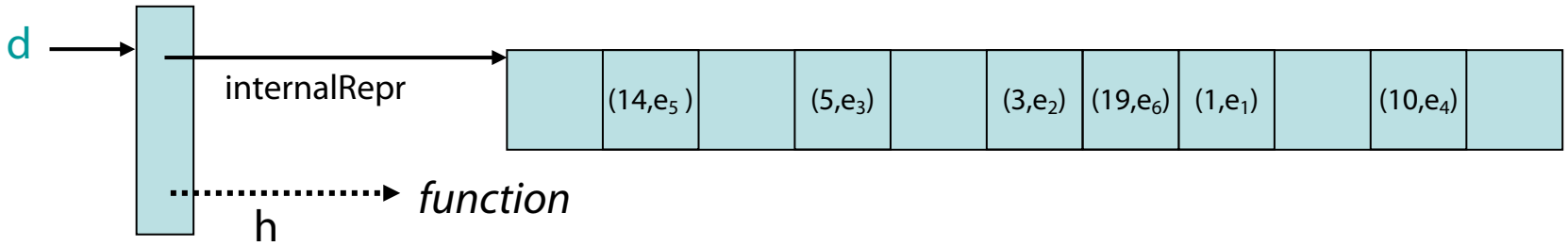
procedure insert(k, e, d:Dictionary)
  T := internalRepr(d)
  T[h(k)] := (k,e)

Parameter (d) kann weggelassen werden (wie bei h)

procedure insert(k, e, d:Dictionary)
  internalRepr[h(k)] := (k,e)

Mehrdeutigkeiten möglich?

procedure insert(k:Integer, e:Type42, d:Dictionary)
  internalRepr(d)[h(k)] := (k,e)



Hashing: Übliches Anwendungsszenario

- Menge U der potentiellen Schlüssel ist „groß“
- Anzahl der Feldelemente length(T) „klein“
- D.h.: |U| >> length(T), aber nur „wenige“ u ∈ U werden tatsächlich betrachtet
- Werte u können auch „groß“ sein (viele Bits)
  - Große Zahlen, Tupel mit vielen Komponenten, Bäume, ...
  - Eventuell nur Teile von u zur einfachen Bestimmung des Index für T betrachtet
    - Nur einige Zeichen einer Zeichenkette betrachtet
    - Bäume nur bis zu best. Tiefe betrachtet
  - Sonst Abbildungsvorgang h evtl. zu aufwendig
- Verschiedene Schlüssel möglicherweise auf gleichen Index abgebildet (Kollision)

Hashfunktionen

- Hashfunktionen müssen i.A. anwendungsspezifisch definiert werden (oft für Basisdatentypen Standardimplementierungen angeboten)
- Hashwerte sollen möglichst gleichmäßig gestreut werden (sonst Kollisionen vorprogrammiert)
- Ein erstes Beispiel für U = Integer:

```
function h(u)
    return u mod m
```

wobei m=length(T)

Falls m keine Primzahl:
Schlüssel seien alle Vielfache von 10 und Tabellengröße m sei 100
→ Viele Kollisionen

Hashing zur Assoziation und zum Suchen

Analyse bei perfekter Streuung

- insert: O(f(u, h)) mit f(u, h) = 1
  für u∈Integer und h hinreichend einfach
- delete: O(f(u, h)) dito
- lookup: O(f(u, h)) dito

Problem: perfekte Streuung
Sogar ein Problem: gute Streuung

Fälle:

- Statisches Wörterbuch: nur lookup
- Dynamisches Wörterbuch: insert, delete und lookup

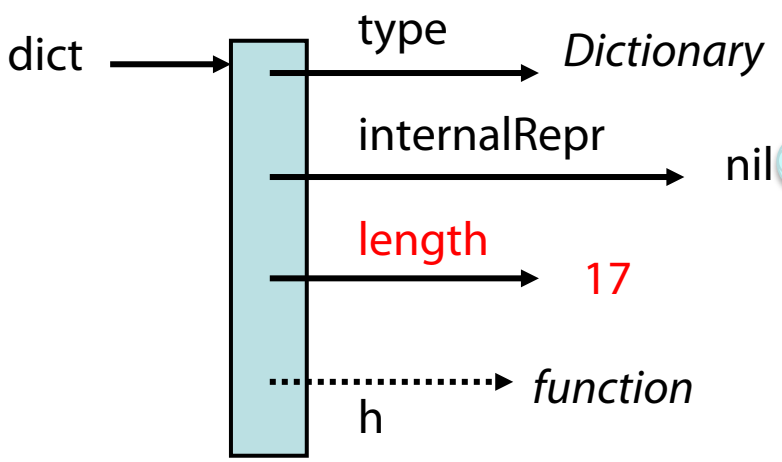
Hashfunktionen

- Beim Erzeugen eines Wörterbuchs kann man die initiale Größe der Hashtabelle angeben

type Dictionary () (internalRepr, length)

method h(d:Dictionary)(u:Integer) **return** u mod length(d)

dict := <(1, “eins”), (5, “fünf”), (42, “zweiundvierzig”)>:Dictionary with length as 17



Annahme:
In initializeInstance
wird für Instanzen von
Dictionary standardmäßig ein
Array der Länge length als
Hashtabelle erzeugt

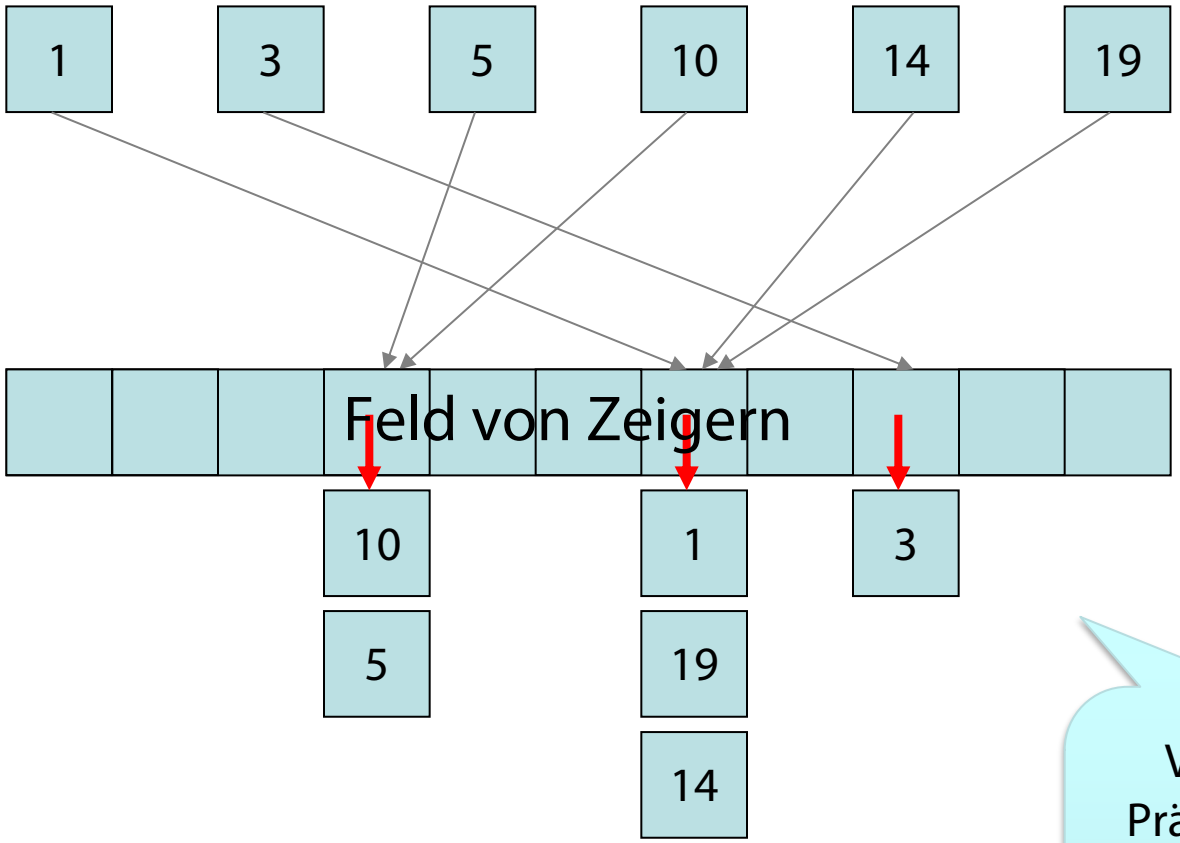
Hashfunktionen

- Man möchte den Nutzer nicht zwingen, sich für length eine Primzahl auszudenken
- Veränderte Hashfunktion für Integer:

```
method h(d:Dictionary)(u:Integer)
  return (u mod p) mod length(d)
```

wobei p> length(d) eine „interne“ Primzahl und length(d) nicht notwendigerweise prim

Hashing mit Verkettung[1] (Kollisionslisten)



unsortierte verkettete Listen

Vereinfachte Präsentation der Tupel (nur Schlüssel dargestellt)

[1] Auch geschlossene Adressierung genannt.

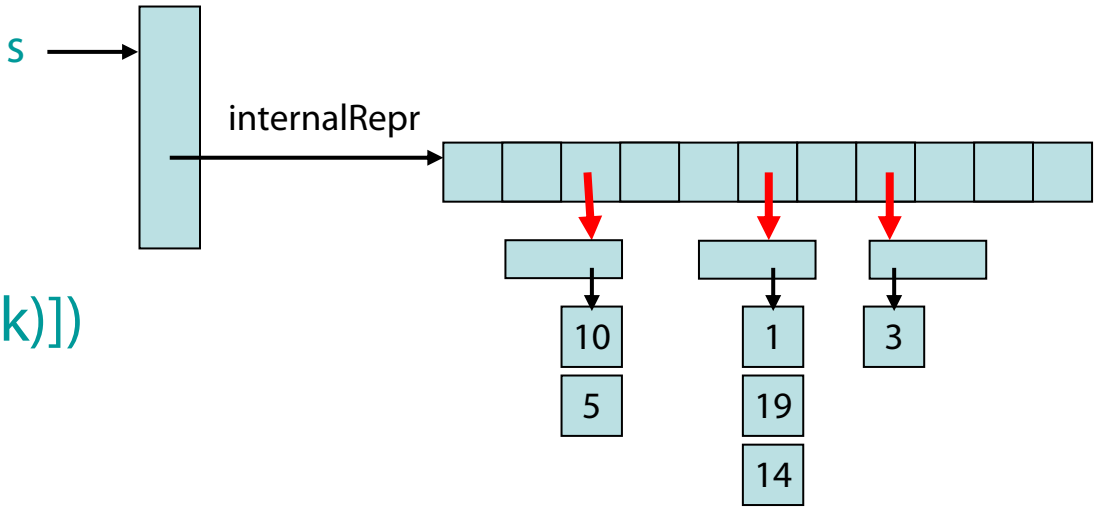
Hashing mit Verkettung

T: Array [0..m-1] of List

procedure insert(k, e, d:):
    T := internalRepr(d);
    insert_in_list((k, e), T[h(k)])

procedure delete(k, d:):
    T := internalRepr(d);
    delete_from_list((k, e), T[h(k)])

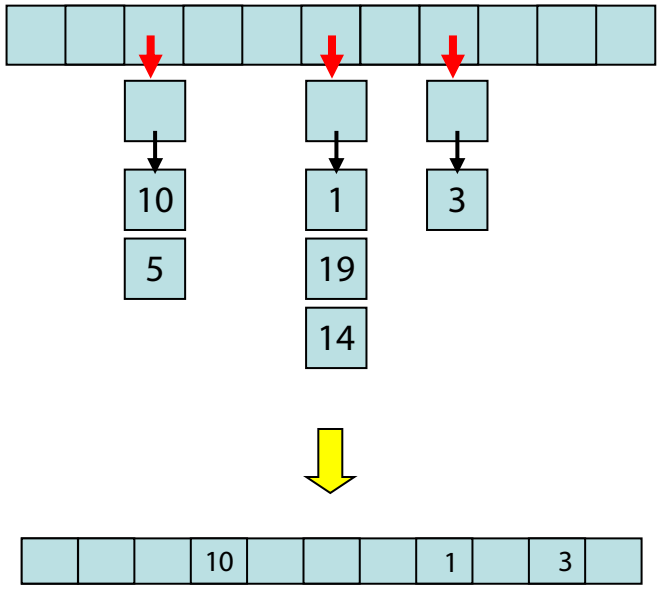
function lookup(k, d:):
    T := internalRepr(d)
    for (k‘, e) ∈ T[h(k)] **do**
        if k=k‘ **then return** e
    return ⊥



Analyse der Komplexität bei Verkettung

- Sei α die durchschnittliche Länge der Listen, dann
  - $\Theta(1+\alpha)$ für
    - erfolglose Suche und
    - Einfügen (erfordert Überprüfung, ob Element schon eingefügt ist)
  - $O(1+\alpha)$ für erfolgreiche Suche

Kollisionslisten im Folgenden nur durch das erste Element direkt im Feld dargestellt



Dynamische Hashtabelle

Problem: Hashtabelle kann zu groß oder zu klein sein

Lösung: Reallokation

- Wähle neue geeignete Tabellengröße

Strategie:
Verdopplung bzw. Halbierung
(Faktor könnte bei der Erzeugung einer Hashtabelle angegeben werden)

- Wähle ggf. neue Hashfunktion
- Übertrage Elemente auf die neue Tabelle
  - Jeweils mit Anwendung der (neuen) Hashfunktion
  - In den folgenden Darstellung ist dieses nicht gezeigt!

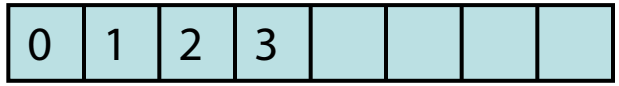
Dynamische Hashtabelle

Wegen Kollisionen evtl. für Hashtabelle schon ab n>m/2 nötig

- Sei m die Größe des Feldes, n die Anzahl der Elemente
- Tabellenverdopplung (n>m):

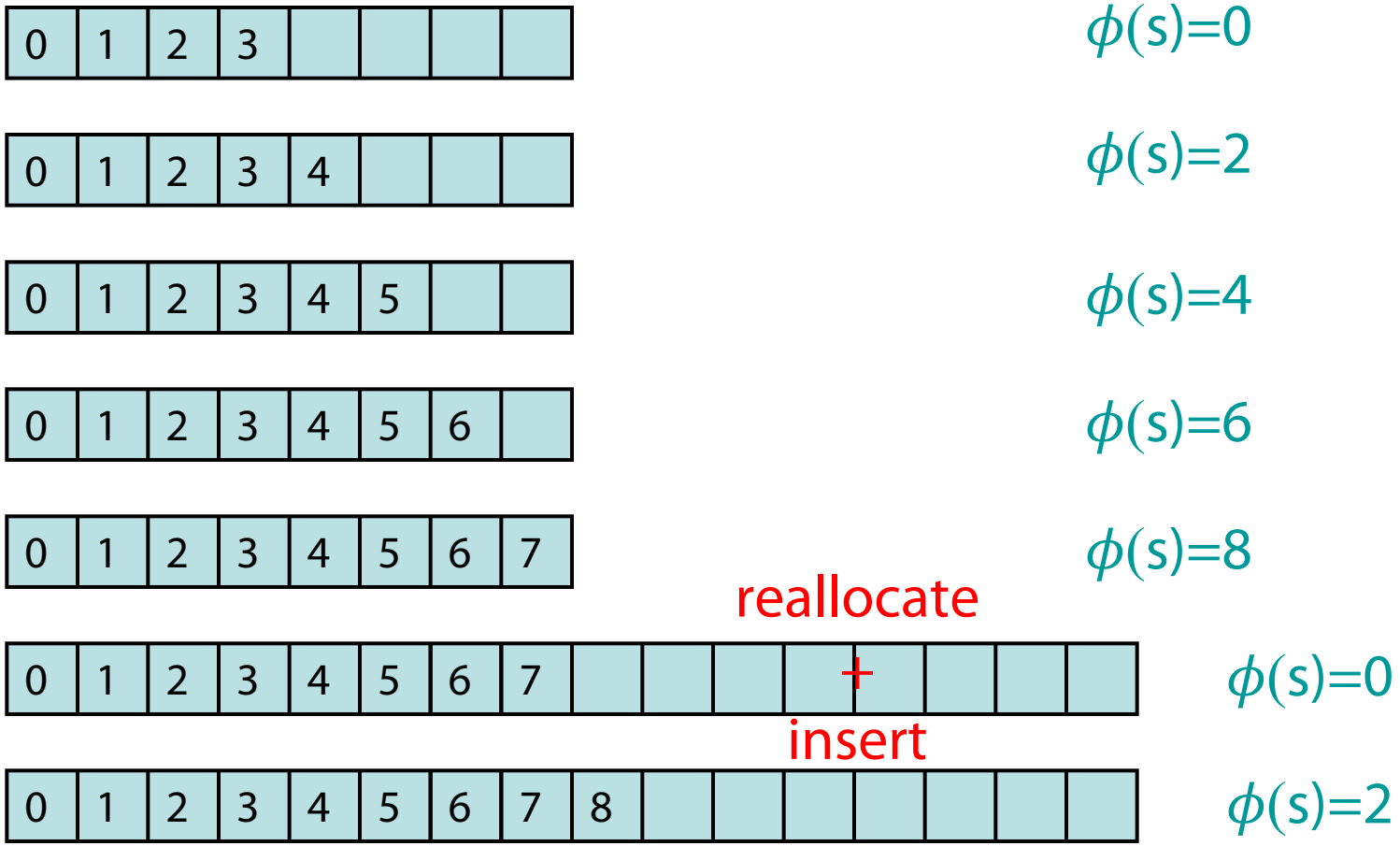
- Tabellenhalbierung (n≤m/4):

- Von

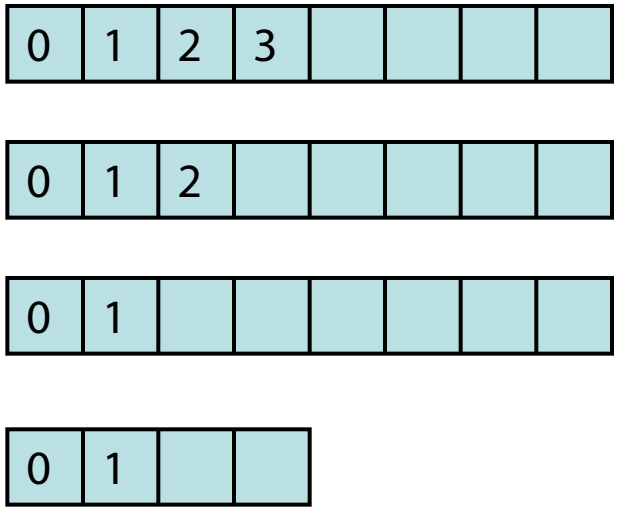


  - Nächste Verdopplung: > n insert Ops
  - Nächste Halbierung: > n/2 delete Ops

Dynamische Hashtabelle: Potential



Dynamische Hashtabelle: Potential



$\phi(s)=0$

$\phi(s)=2$

$\phi(s)=4$

$\phi(s)=0$

Generelle Formel für $\phi(s)$:
(w_s: Feldgröße in s, n_s: Anzahl Einträge)

$$\phi(s) = 2|w_s/2 - n_s|$$

Dynamische Hashtabelle: Potential

Generelle Formel für $\phi(s)$:
(w$_s$: Feldgröße von s, n$_s$: Anzahl Einträge)

$$\phi(s) = 2|w_s/2 - n_s|$$

Behauptung:
Sei $\Delta\phi = \phi(s') - \phi(s)$ für $s \longrightarrow s'$. Für die amortisierten Laufzeiten gilt:

- insert: $t_{ins} + \Delta\phi$ insert $\in O(1)$ amort.
- delete: $t_{del} + \Delta\phi$ delete $\in O(1)$ amort.

Hashtabellen können „ruckeln“

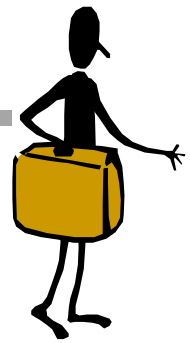
Dynamische Hashtabelle: Primzahlbestimmung

Problem: Tabellengröße m sollte prim sein
(für gute Verteilung der Schlüssel)
Wie finden wir Primzahlen?

Lösung:

- Für jedes k gibt es Primzahl in $[k^3,(k+1)^3]$
- Wähle Primzahlen m, so dass $m \in [k^3, (k+1)^3]$
- Jede nichtprime Zahl in $[k^3,(k+1)^3]$ muss Teiler $< \sqrt{(k+1)^3}$ haben
  $\longrightarrow$ erlaubt effiziente Primzahlfindung

Zusammenfassung



- Assoziationen Schlüssel -> Objekt effektiv verwalten
- Wir müssen nicht für jede Assoziation in den Instanzen eine Komponente vorsehen
- Wichtige Grundlage des Software-Engineering
  - Modellierung der Objekte aus Sicht der Anwendung
  - Wenn mit Objekten in bestimmten Kontexten bestimmte Information assoziiert werden soll, verwende Hash-Tabellen in dem Kontext
  - Man muss die Modellierung nicht auf die jeweilige Verwendung abstimmen (Glass-Box-Datenstrukturen)
  - Man kann auch nicht jede Verwendung antizipieren