Algorithmen und Datenstrukturen

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Magnus Bender und Malte Luttermann
(Übungen)

sowie viele Tutoren


Assoziation von Schlüssel zu Objekten
Wörterbücher und Hashing
C4 EAST OF ENGLAND
C5 NOTTINGHAM
STOCKPORT SK ALL
BA2.2 BATH
SWINDON SN ALL
C6 MANCHESTER M1-4/60 & 99
D6 BRIDGWATER TA5-7
E6 BATH BA1-2
B6 L RES (EXC41-75 & GIRO)
D7 TAUNTON TA1-4
C7 TA RESIDUE
B7 STEVENAGE SG ALL
B7 WATCHET TA23

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

Wörterbuch-Datenstruktur

s: Menge von Schlüssel-Wert-Paaren (key, object)

Operationen:

- insert(k, e, s): $s = s \cup \{(k, e)\}$
  // Änderung nach außen sichtbar
- delete(k, s): $s = s \setminus \{(k, e)\}$, wobei e das Element ist, das unter dem Schlüssel k eingetragen ist
  // Änderung von s nach außen sichtbar
- lookup(k, s): Falls es ein $(k, e) \in s$ gibt, dann gib e aus, sonst gib $\perp$ (bzw. nothing) aus

Wörterbücher

Unterschied 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)

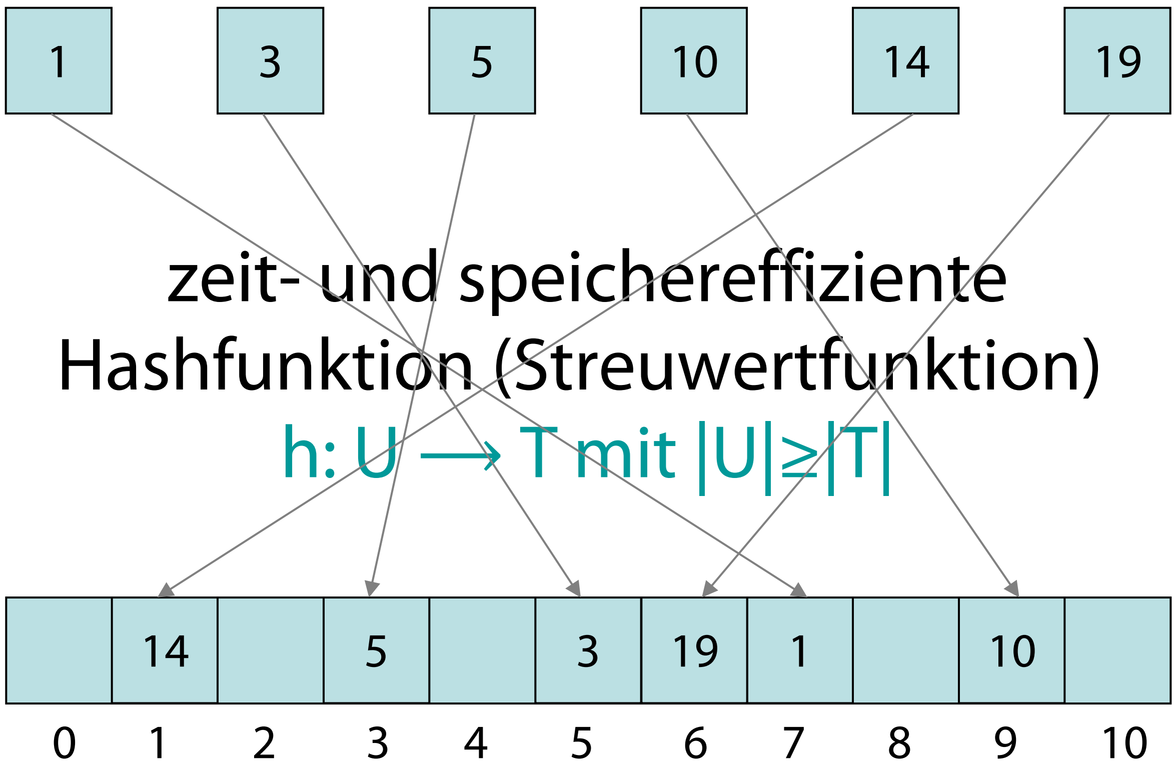
    ```
    for k in keys(dict)    ... end
    ```
  - Über alle Werte

    ```
    for v in values(dict)    ... end
    ```
  - Über alle Attribut-Wert-Paare

    ```
    for (k, v) in pairs(dict)    ... end
    ```

Hashing (Streuung)

Einige Elemente
aus einer Menge U:

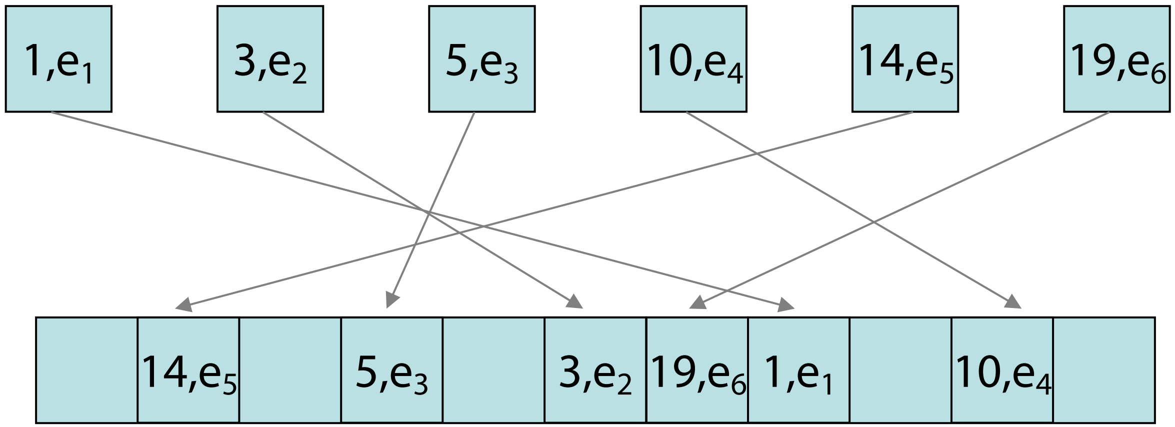


zeit- und speichereffiziente
Hashfunktion (Streuwertfunktion)
$h: U \longrightarrow T$ mit $|U| \geq |T|$

Hashtabelle T

Assoziation durch Hashing

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



- Schlüssel k selbst können auch Objekte sein, es muss nur eine Abbildung auf T definiert sein bzw. werden

Hashing: Übliches Anwendungsszenario

- Menge U der potentiellen Schlüssel u „groß“
- Anzahl der Feldelemente length(T) „klein“
- D.h.: |U| >> length(T), aber nur „wenige“ u ∈ U werden tatsächlich betrachtet
- Werte u können „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
- Folge der großen Menge bzw. der teilweisen Betrachtung:
  - Verschiedene Elemente 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 % m # modulo
end
```

  wobei m die Länge des Feldes ist

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

Warum i.A. nicht?

- Kann man h auf komplexen Objekten über deren „Adresse" realisieren?

Hashfunktionen

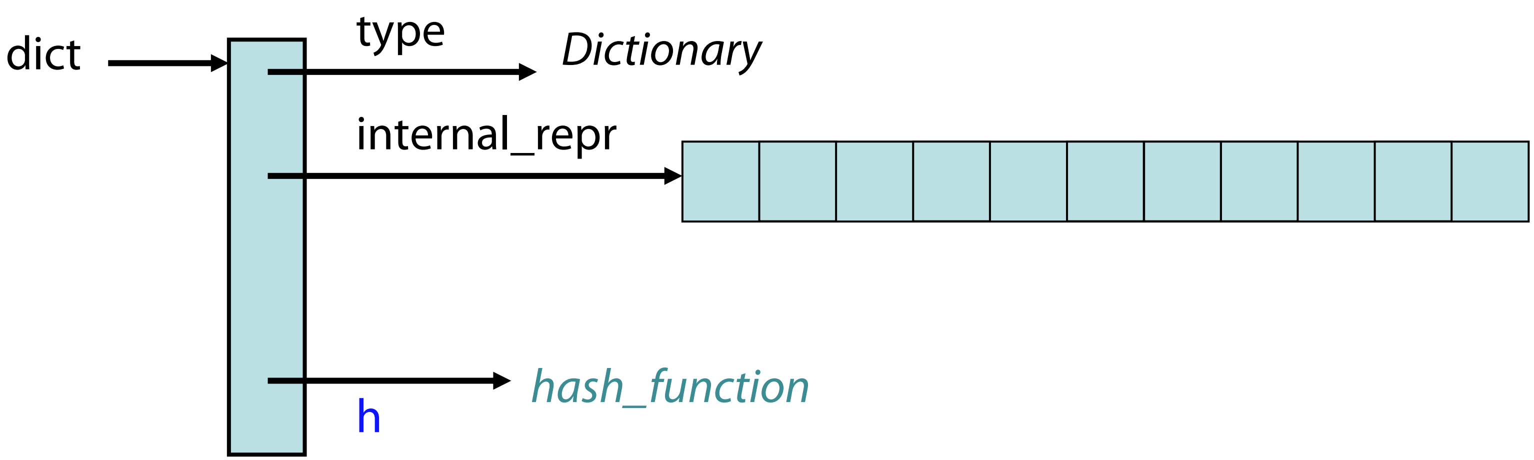
- Man möchte den Nutzer nicht zwingen, sich für length eine Primzahl auszudenken
- Veränderte Hashfunktion für $u \in$ Integer :

```
function h(u)
   return ( u % p ) % m
end
```

wobei $p > m$ eine „interne“ Primzahl und m nicht notwendigerweise prim

Dictionary selbst gebaut

```
struct Dictionary
  internal_repr :: Array{Any}
  h :: Function
end
```

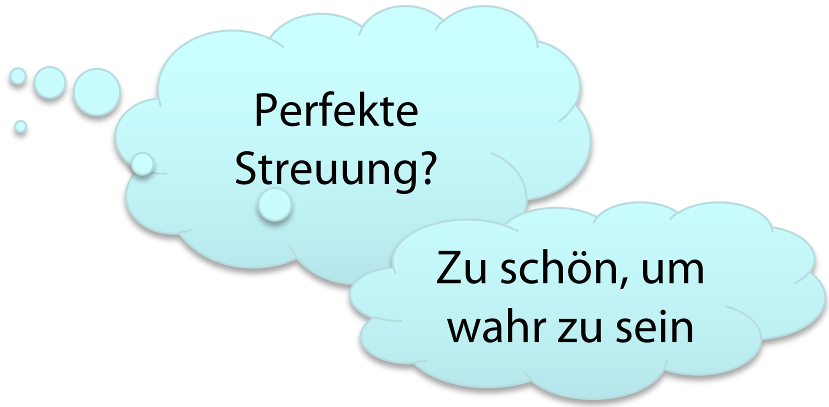


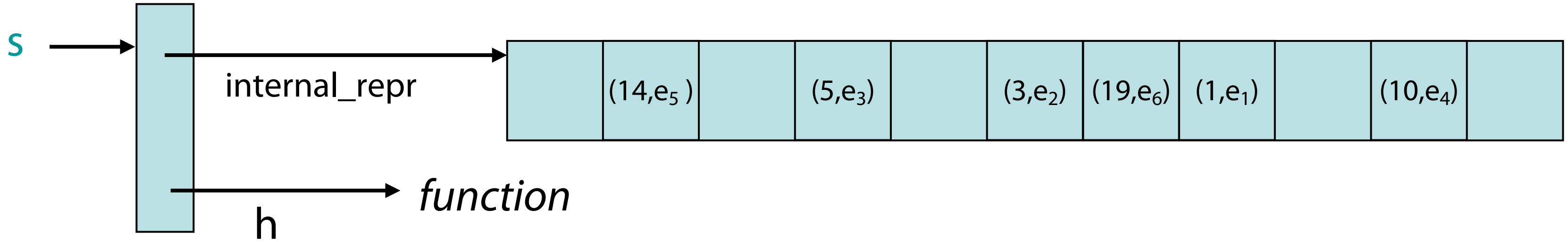

```
function initialize_dictionary( init_values, length, map_to_int )
    p = larger_prime(length)
    h = (x)->(map_to_int(x) % p) % length + 1
    d = Dictionary( Array{Any}(nothing,length), h )
    for (k, e) in init_values
        insert(k, e, d)
    end
    return d
end

d = initialize_dictionary( [...], 100, my_map_to_int)
```

Hashing (perfekte Streuung, keine Kollisionen)

```
function insert(k, e, d)
  T = d.internal_repr
  T[d.h(k)] = (k, e)
end
function delete(k, d)
  T = d.internal_repr
  T[d.h(k)] = nothing
end
function lookup(k, d)
  T = d.internal_repr
  t = T[d.h(k)]
  return if isnothing(t)  nothing else t[2] end
end
```





Hashing zur Assoziation und zum Suchen

Analyse bei perfekter Streuung

- insert: O(f(map_to_int, h)) = O(1)
  für map_to_int hinreichend einfach
- delete: O(f(map_to_int, h)) dito
- lookup: O(f(map_to_int, h)) dito

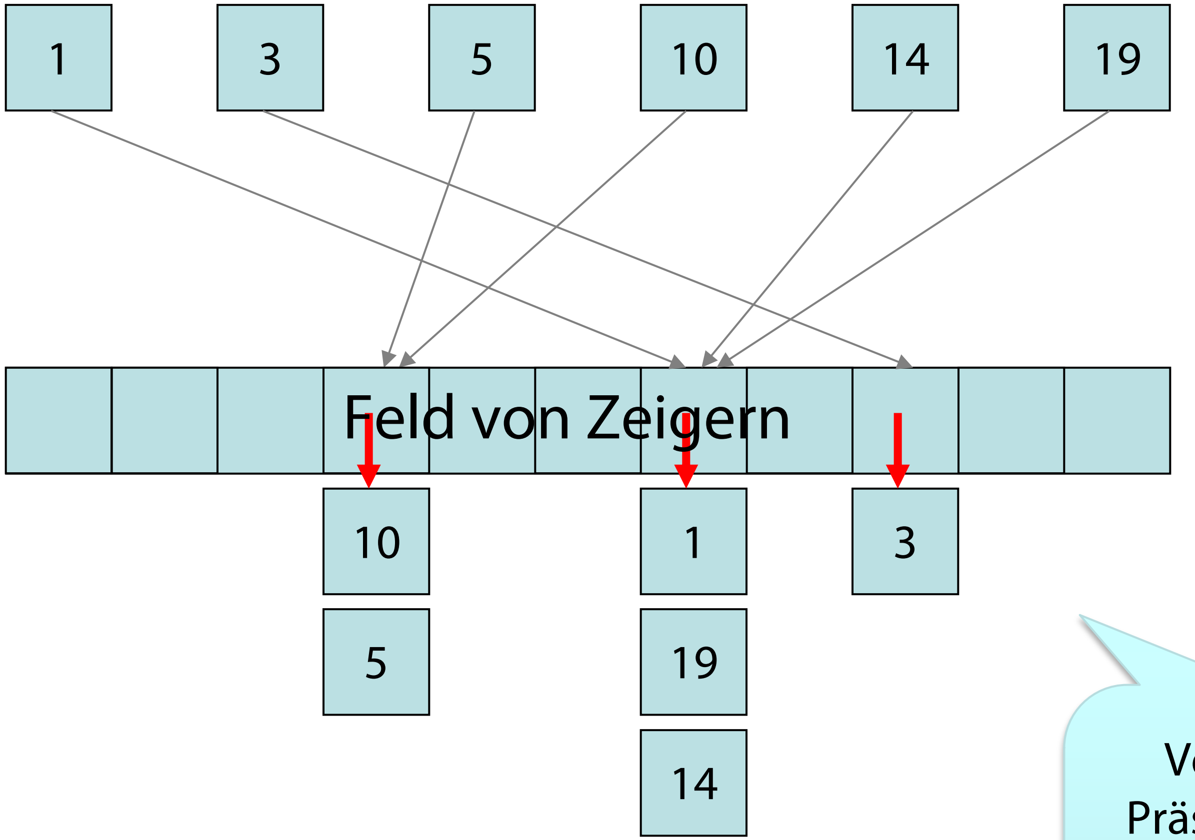
Problem: perfekte Streuung

Sogar ein Problem: gute Streuung

Fälle:

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

Hashing mit Verkettung[1] (Kollisionslisten)



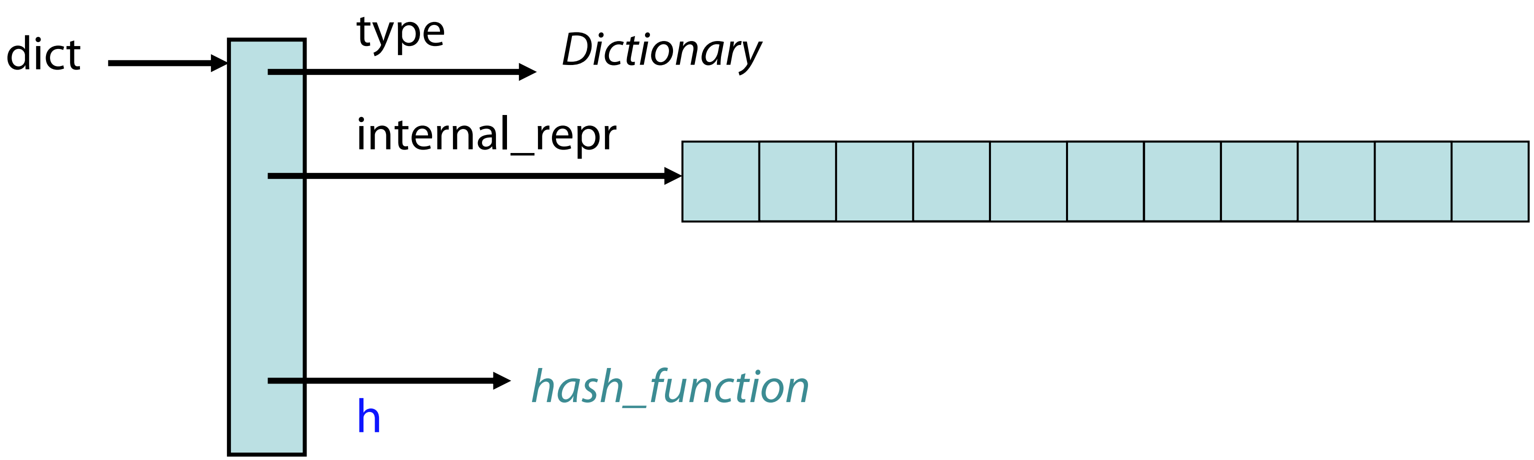
unsortierte verkettete Listen

Vereinfachte Präsentation der Tupel (nur Schlüssel dargestellt)

[1] Auch geschlossene Adressierung genannt.

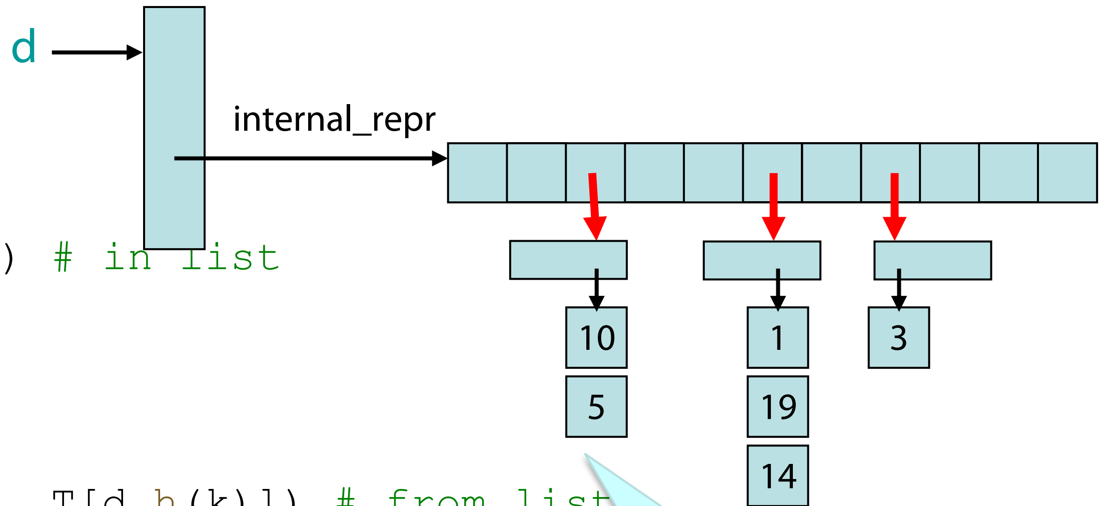
Dictionary selbst gebaut

```
struct Dictionary
  internal_repr :: Array{Any}
  h :: Function
end
```




```
function initialize_dictionary( init_values, length, map_to_int )
    p = larger_prime(length)
    h = (x)->(map_to_int(x) % p) % length + 1
    d = Dictionary( Array{Any}(nothing,length), h )
    for i = 1:length
        d.internal_repr[i]=make_list()
    end
    for (k, e) in init_values
        insert(k, e, d)
    end
    return d
end
```

Hashing mit Verkettung

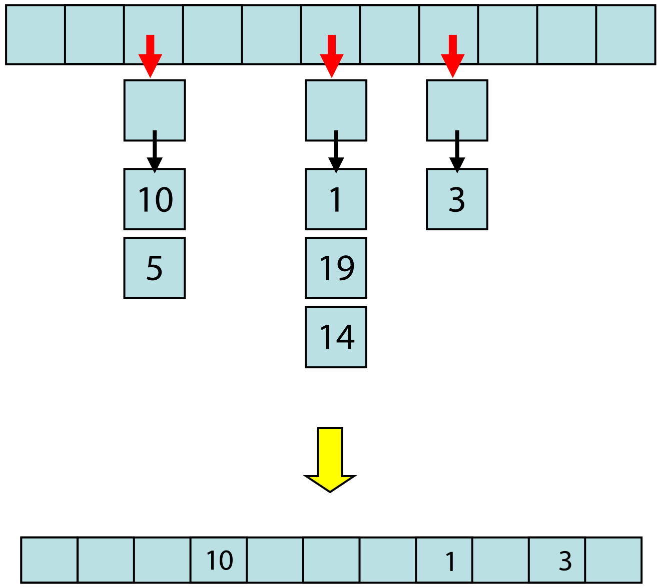



```
function insert(k, e, d)
  T = d.internal_repr
  insert((k, e), T[d.h(k)]) # in list
end
function delete(k, d)
  T = d.internal_repr
  delete((k, lookup(k, d)), T[d.h(k)]) # from list
end
function lookup(k, d)
  T = d.internal_repr
  for (k_, e) in T[d.h(k)]
    if k == k_    return e end
  end
  return nothing
end
```

Vereinfachte Präsentation der Tupel (nur Schlüssel dargestellt)

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

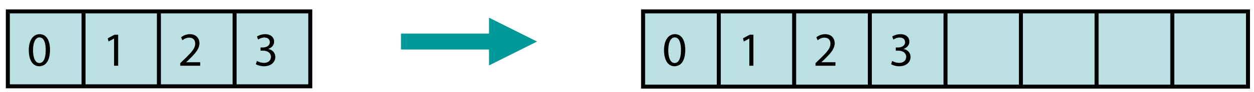
Lösung: Reallokation

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

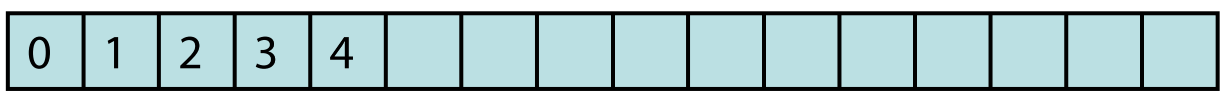
Dynamische Hashtabelle

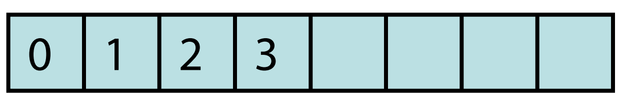
Wegen Kollisionen evtl. für Hashtabelle schon ab n>m/2 nötig

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



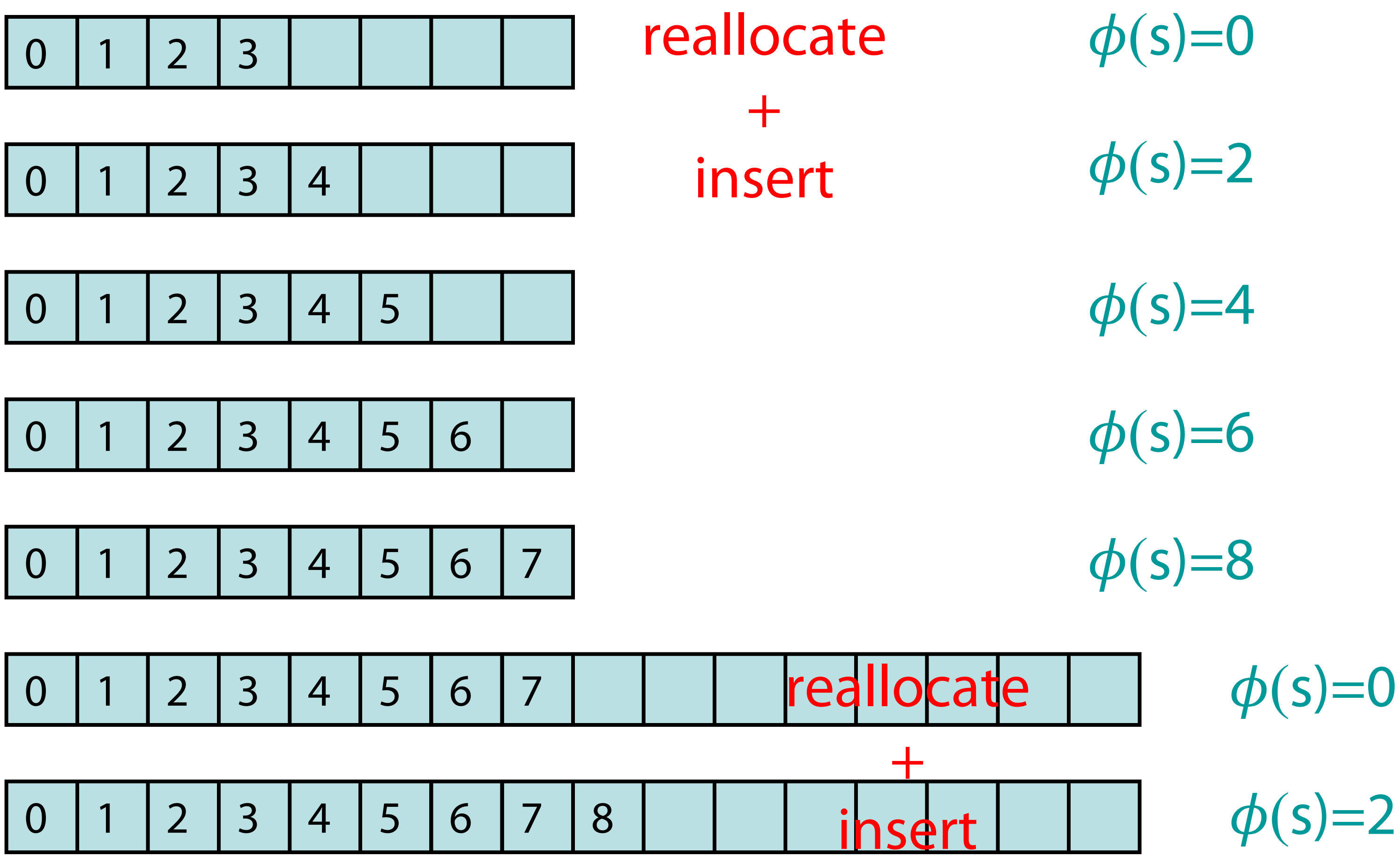
- Tabellenhalbierung (n≤m/4):



- Von 

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

Dynamische Hashtabelle



Dynamische Hashtabelle

0	1	2	3						$\phi(s)=0$
0	1	2							$\phi(s)=2$
0	1							delete	$\phi(s)=4$
								+	
0	1							reallocate	$\phi(s)=0$

Generelle Formel für $\phi(s)$:

(w_s: Feldgröße von s, n_s: Anzahl Einträge)

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

Dynamische Hashtabelle

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

Dynamische Hashtabelle

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

Offene Adressierung

- Bei Kollision speichere das Element „woanders" in der Hashtabelle
- Vorteile gegenüber Verkettung
  - Keine Verzeigerung
  - Schneller, da Speicherallokation für Zeiger relativ langsam
- Nachteile
  - Langsamer bei Einfügungen
    - Eventuell sind mehrere Versuche notwendig, bis ein freier Platz in der Hashtabelle gefunden worden ist (Sondierung)
  - Tabelle muss größer sein (maximaler Füllfaktor kleiner) als bei Verkettung, um Effektivität bei den Basisoperationen zu erreichen

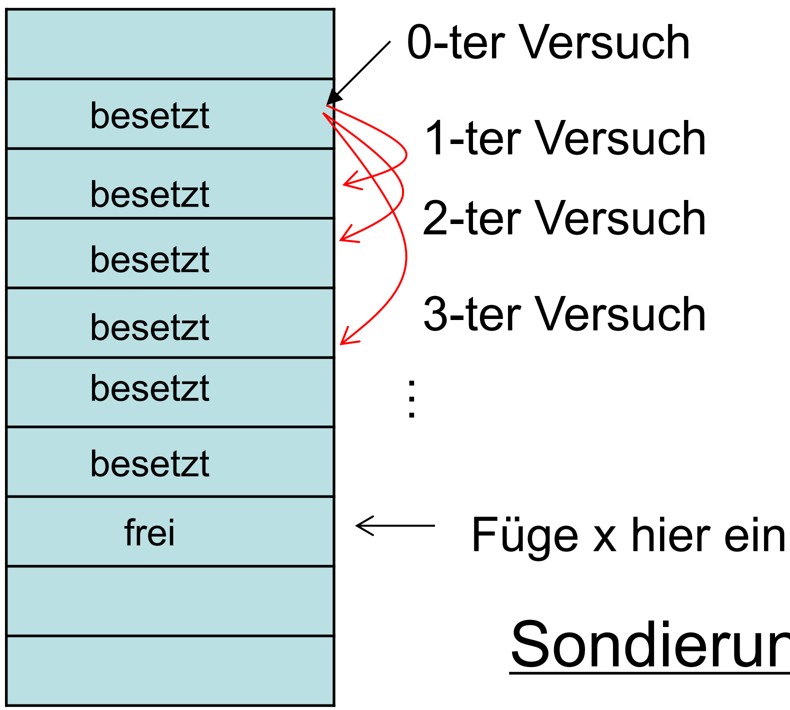
Offene Adressierung

- Eine Sondierungssequenz ist eine Sequenz von Indizes in der Hashtabelle für die Suche nach einem Element
  - $h_0(x), h_1(x), \ldots$
  - Sollte jeden Tabelleneintrag genau einmal besuchen
  - Sollte wiederholbar sein , …
    - … sodass wir wiederfinden können, was wir eingefügt haben

- Hashfunktion
  - $h_i(x) = (h(x) + f(i)) \bmod m$
  - $f(0) = 0$ Position des 0-ten Versuches
  - $f(i)$ „Distanz des i-ten Versuches relativ zum 0-ten Versuch“

Einfügung von x: Lineares Sondieren

<u>Linear probing:</u>



- f(i) ist eine lineare Funktion von i, z.B. $f(i) = i$

i-ter Versuchs-index

0-ter Versuchs-index

+ i

$$h_i(x) = (h(x) + i) \bmod m$$

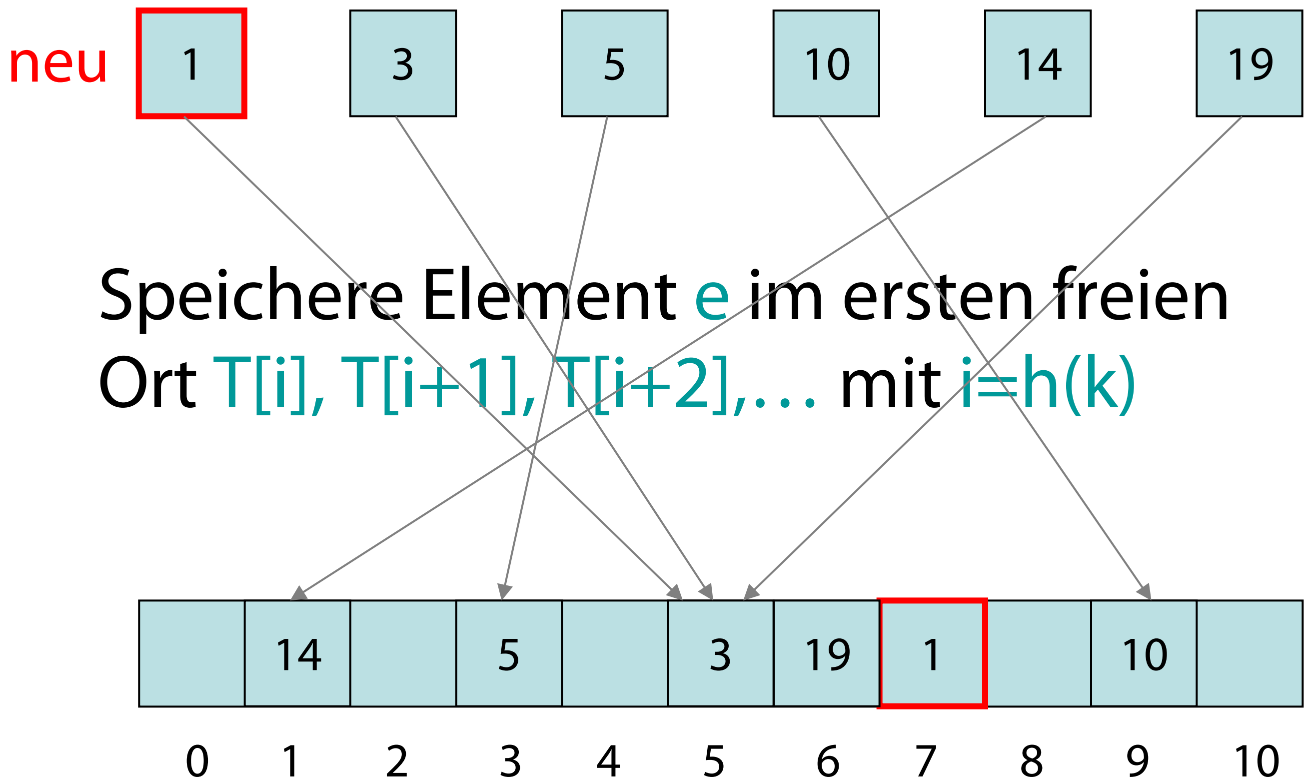
<u>Sondierungssequenz:</u> +0, +1, +2, +3, +4, …

Fahre fort bis ein freier Platz gefunden ist

#fehlgeschlagene Versuche als eine Messgröße der Performanz

Hashing mit Linearer Sondierung (Linear Probing)

Speichere Element e im ersten freien Ort T[i], T[i+1], T[i+2],… mit i=h(k)



Hashing mit Linearer Sondierung

T: Array [1..m] of pairs (key, element)   // m>n

```
function insert(k, e, d)
    T = d.internal_repr; i = d.h(k)
    while !isnothing(T[i]) && T[i][1] != k
        i = (i + 1) % length(T)
    end
    T[i] = (k,e)
end

function lookup(k, d)
    T = d.internal_repr; i = d.h(k)
    while !isnothing(T[i]) && T[i][1] != k
        i = (i + 1) % length(T)
    end
    if isnothing(T[i]) || T[i][1] != k
        return nothing
    else
        return T[i][2]
    end
end
```

Hashing mit Linearer Sondierung

Problem: Löschen von Elementen

Lösungen:

1. Verbiete Löschungen
2. Markiere Position als gelöscht mit speziellem Zeichen (ungleich $\perp$)
3. Stelle die folgende Invariante sicher:
   Für jedes $e \in S$ mit idealer Position $i=h(k)$ und aktueller Position j gilt

$$T[i],T[i+1]\ldots,T[j] \text{ sind besetzt}$$

Nachteile der Linearen Sondierung

- Sondierungssequenzen werden mit der Zeit länger
  - Schlüssel tendieren zur Häufung in einem Teil der Tabelle
  - Schlüssel, die in den Cluster gehasht werden,
    am Ende des Clusters gespeichert
    (→ vergrößern damit den Cluster)
  - Seiteneffekt
    - Andere Schlüssel sind auch betroffen, falls sie in die Nachbarschaft gehasht werden

Analyse der offenen Adressierung

- Sei $\alpha=n/m$ mit n Anzahl eingefügter Elemente und m Größe der Hashtabelle
  - α wird auch ***Füllfaktor*** der Hashtabelle genannt
- Anzustreben ist $\alpha\leq 1$
- Unterscheide erfolglose und erfolgreiche Suche

Analyse der erfolglosen Suche

Behauptung: Im typischen Fall $O(1/(1-\alpha))$

- Bei 50% Füllung ca. 2 Sondierungen nötig
- Bei 90% Füllung ca. 10 Sondierungen nötig

Ohne Beweis

Analyse der erfolgreichen Suche

Behauptung: Im durchschnittlichen Fall $O(1/\alpha \ln (1/(1-\alpha)))$

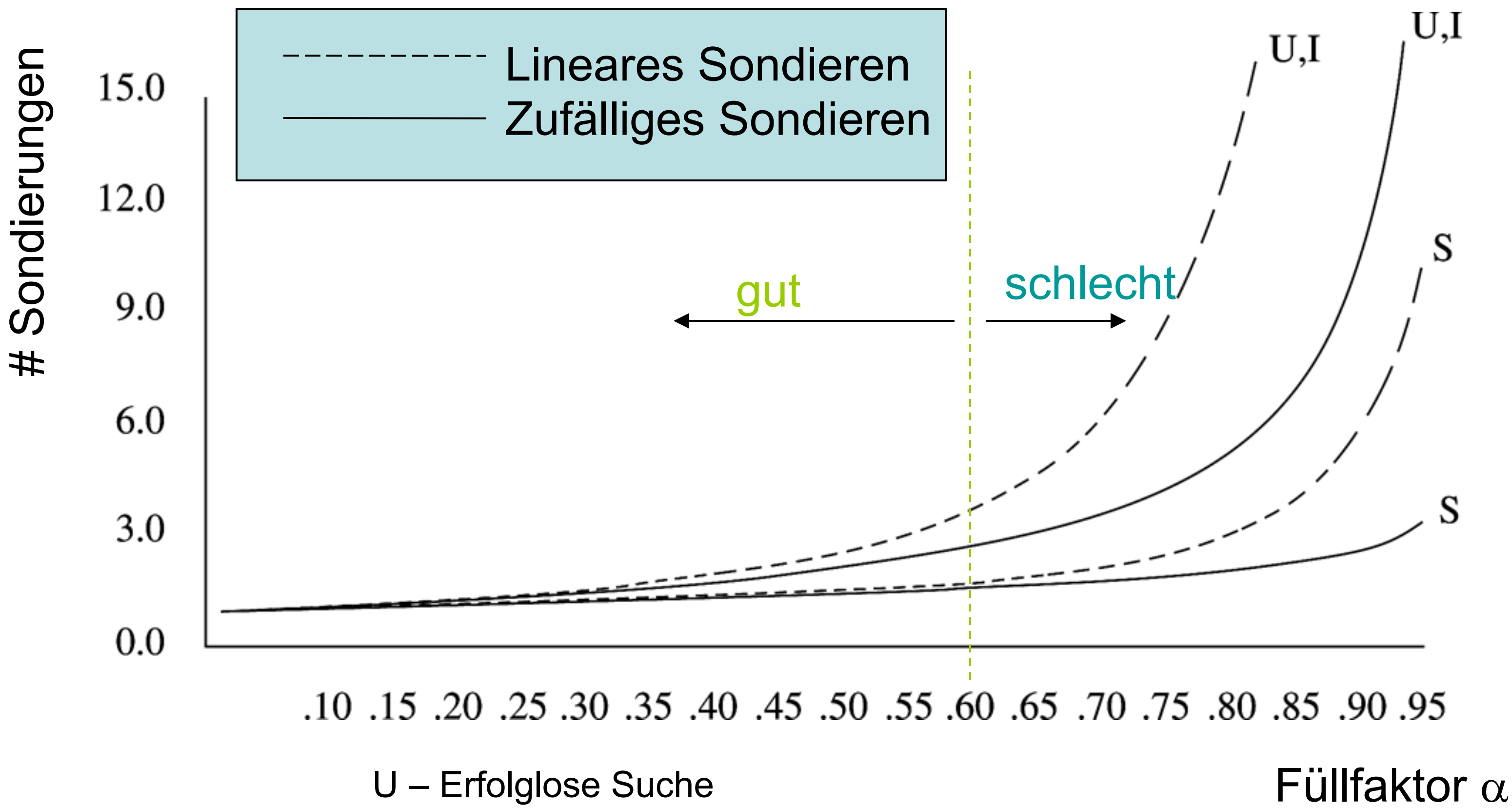
- Bei 50% Füllung ca. 1,39 Sondierungen nötig
- Bei 90% Füllung ca. 2,56 Sondierungen nötig

Ohne Beweis

Zufälliges Sondieren

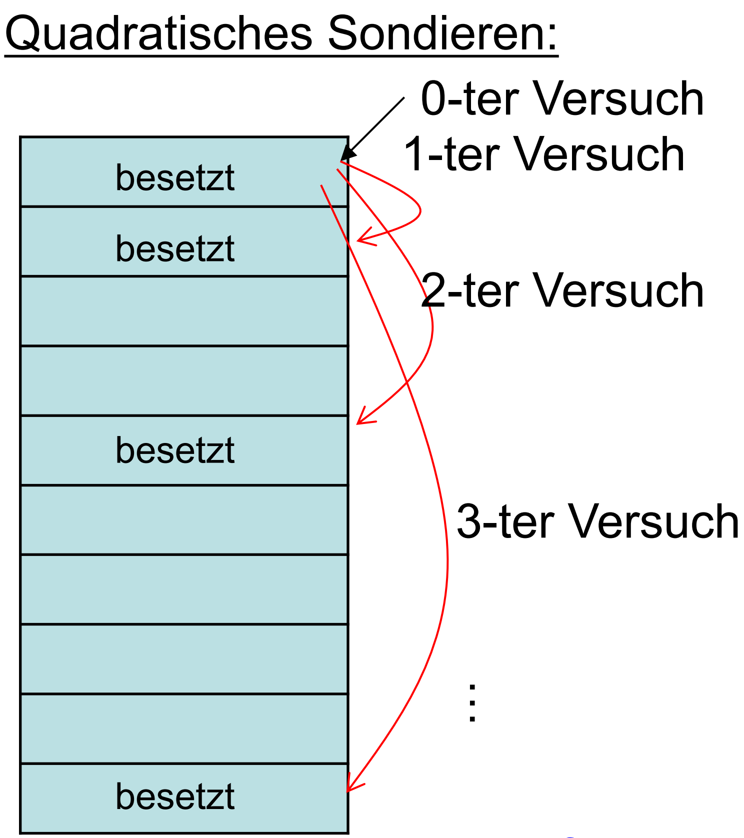
1. Wähle den jeweils nächsten Feldindex nach einer (reproduzierbaren) Zufallsfolge
   - Rechenaufwendig
2. Für jeden Schlüssel k wähle genügend lange zufällige Versatzfolge f(i) und speichere Folge f(i) zur Verwendung bei erneutem Hash von k
   - Speicheraufwendig
   - Bootstrap-Problem
     - Assoziation Key→Indexfolge
     - Realisiert mittels Hashing?

Vergleich mit zufälligem Sondieren



U – Erfolglose Suche
S – Erfolgreiche Suche
I – Einfügen

Quadratisches Sondieren



- Vermeidet primäres Clustering
- $f(i)$ ist quadratisch in i z.B., $f(i) = i^2$
  - $h_i(x) = (h(x) + i^2) \bmod m$
  - Sondierungssequenz: $+0, +1, +4, +9, +16, \ldots$
  - Allgemeiner: $f(i) = c_1 \cdot i + c_2 \cdot i^2$

Fahre fort bis ein freier Platz gefunden ist

#fehlgeschlagene Versuche ist eine Meßgröße für Performanz

Löschen von Einträgen bei offener Adressierung

- Direktes Löschen unterbricht Sondierungskette
- Mögliche Lösung:
  a) Spezieller Eintrag “gelöscht“. Kann zwar wieder belegt werden, unterbricht aber Sondierungsketten nicht. Nachteil bei vielen Löschungen: Lange Sondierungszeiten
  b) Umorganisieren. Kompliziert, sowie hoher Aufwand

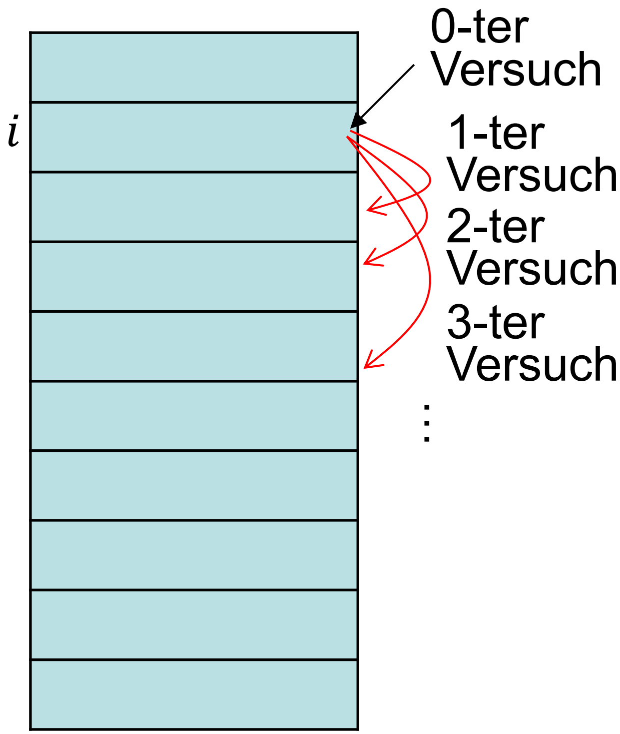
Analyse Quadratisches Sondieren

- Schwierig
- Theorem
  - Wenn die Tabellengröße eine Primzahl ist und der Füllfaktor höchstens ½ ist, dann findet Quadratisches Sondieren immer einen freien Platz
  - Ansonsten kann es sein, dass Quadratisches Sondieren keinen freien Platz findet, obwohl vorhanden
- Damit $\alpha_{max} \leq 1/2$ für quadratisches Sondieren

Review Hashing

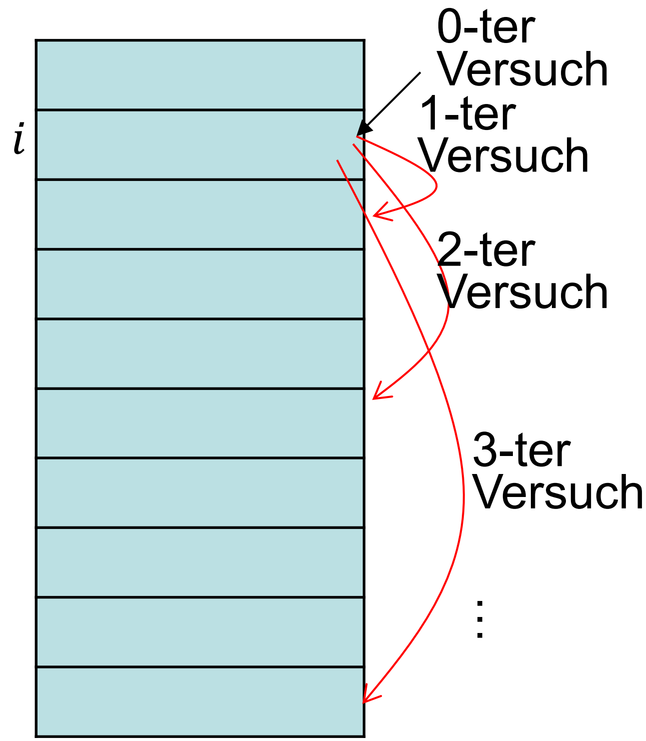
$h_i(x) = (h(x) + f(i)) \bmod m$

<u>Lineares Sondieren:</u>



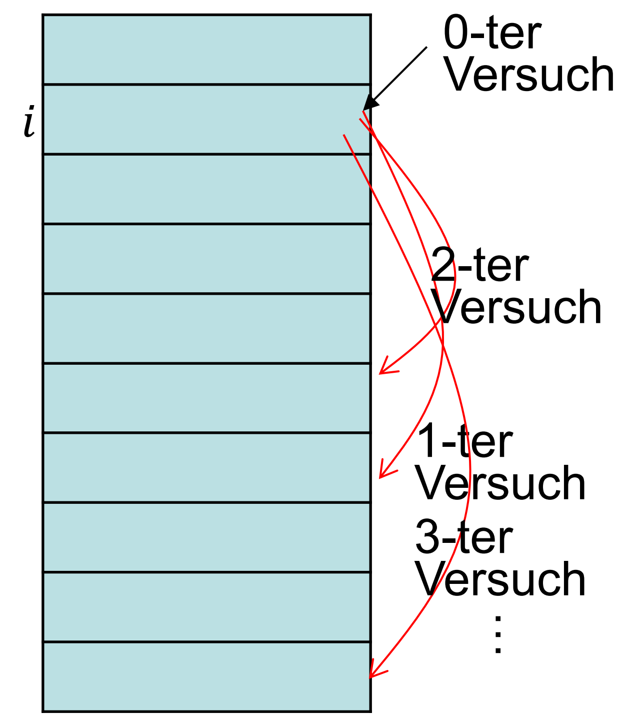
$f(i) = i$

<u>Quadratisches Sondieren:</u>



$f(i) = i^2$

<u>Doppel-Hashing*:</u>



*(bestimmt mit einer zweiten Hashfunktion)

$f(i) = f(i) = i \cdot h'(x)$

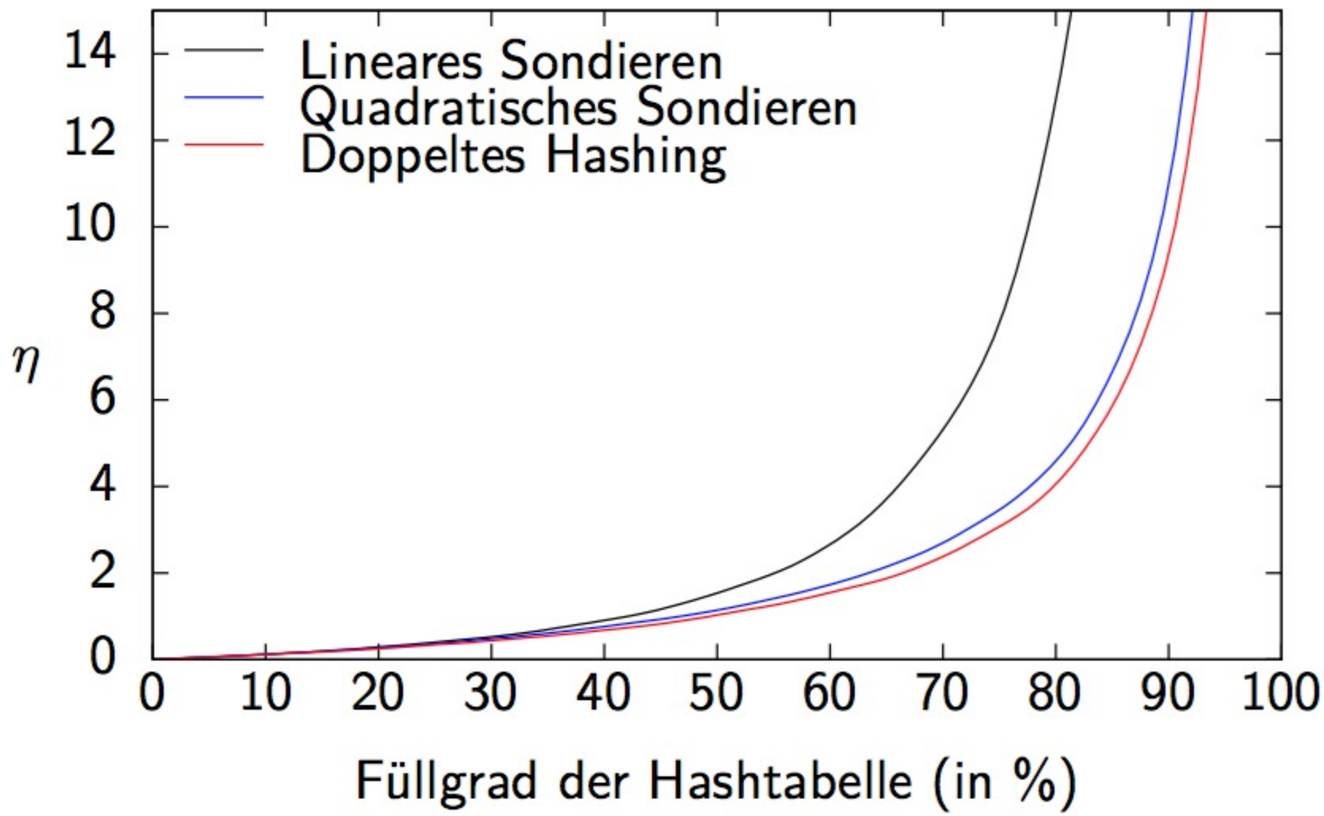
Doppel-Hashing

- Gute Wahl von h'?
  - Sollte niemals 0 ergeben
  - $h'(x) = p - (x \bmod p)$ mit p Primzahl $< m$

L. Guibas and E. Szemerédi: *The Analysis of Double Hashing*, Journal of Computer and System Science, 16, 226-274, **1978**

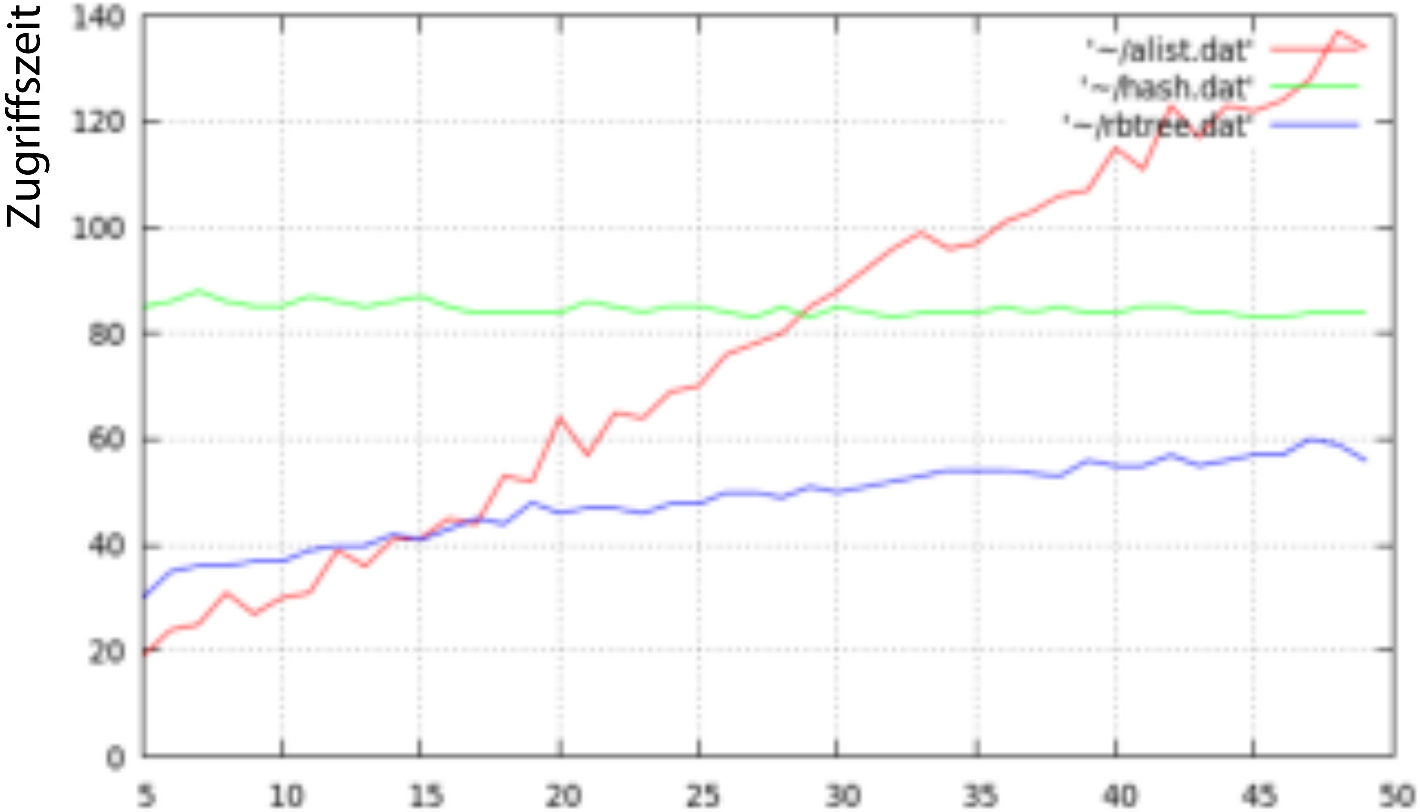
Praktische Effizienz von doppeltem Hashing

- Hashtabelle mit 538 051 Einträgen (Endfüllgrad 99,95%)
- *Mittlere* Anzahl Kollisionen η pro Einfügen in die Hashtabelle:



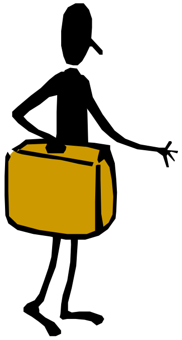
Vergleiche

- Schlüsselwortliste (rot) vs. Hashtabelle (grün) vs. Baum (blau)



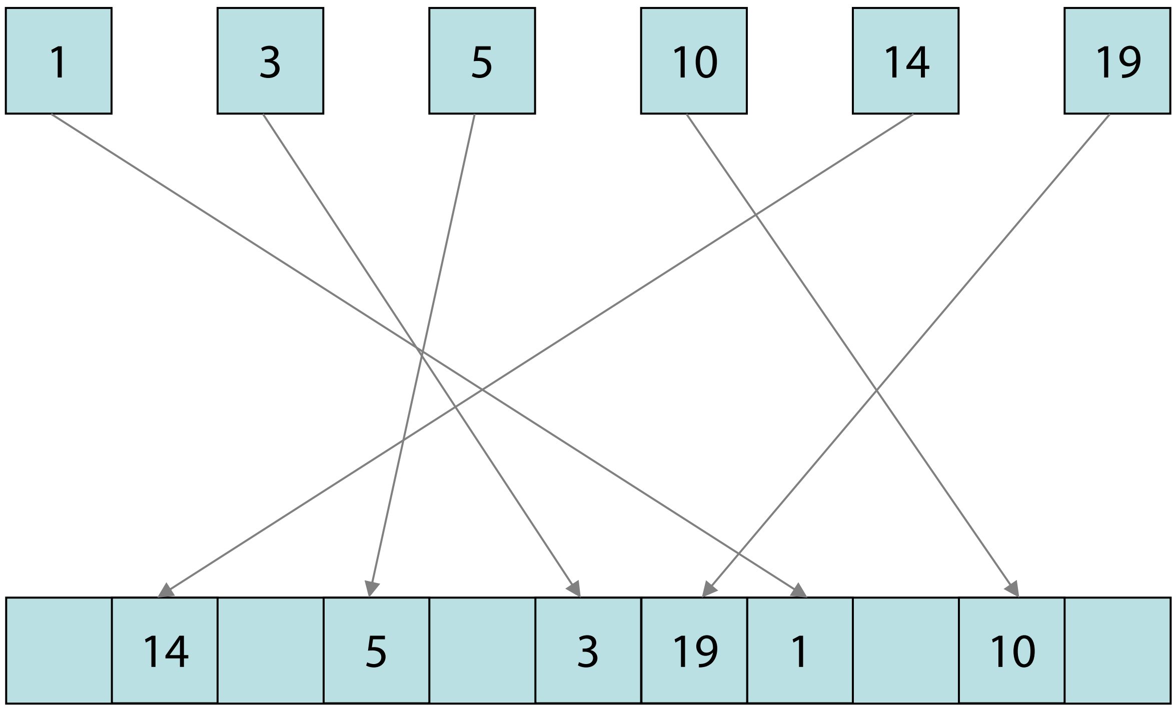
Zusammenfassung: Hashing

- Basisoperation (Suchen, Einfügen, Löschen) in O(1)
- Güte des Hashverfahrens beeinflusst durch
  - Hashfunktion
  - Verfahren zur Kollisionsbehandlung
    - Verkettung
    - Offene Adressierung
      - Lineares/Quadratisches Sondieren/Doppel-Hashing
  - Füllfaktor
    - Dynamisches Wachsen
- Statistisches vs. Dynamisches Hashen
- Universelles Hashing

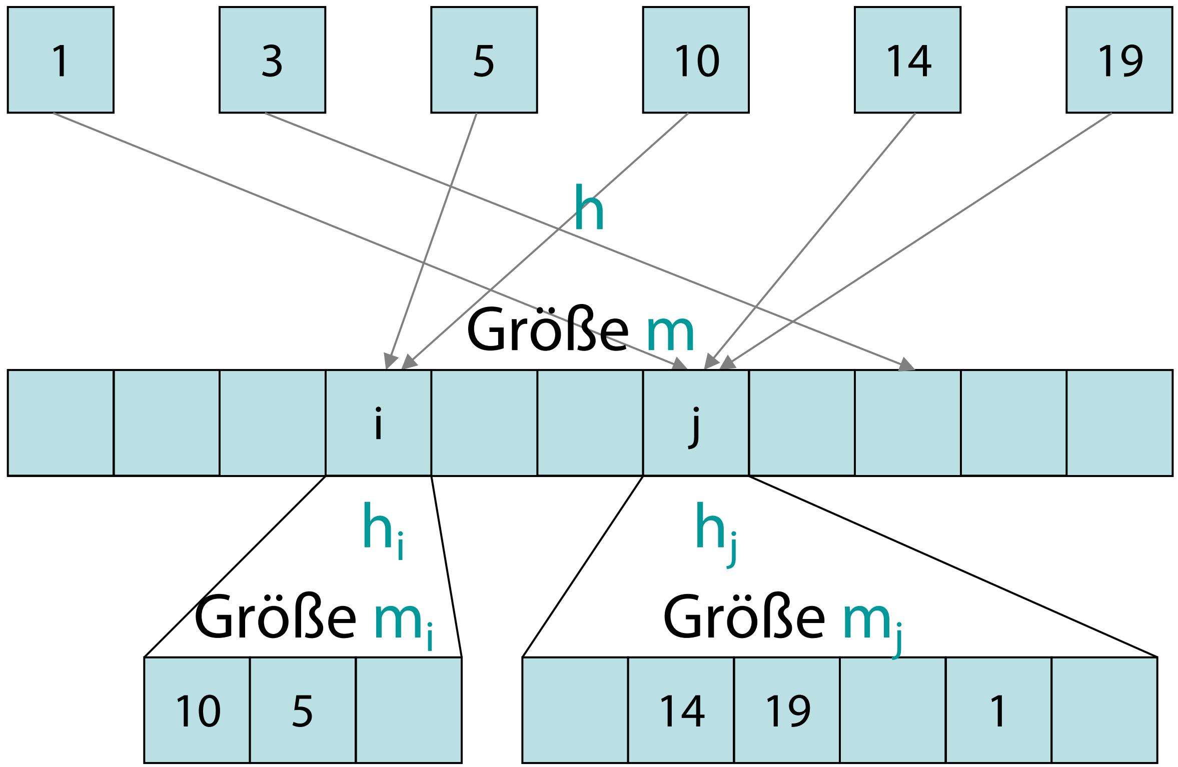


Statisches Wörterbuch

Ziel: perfekte Hashtabelle



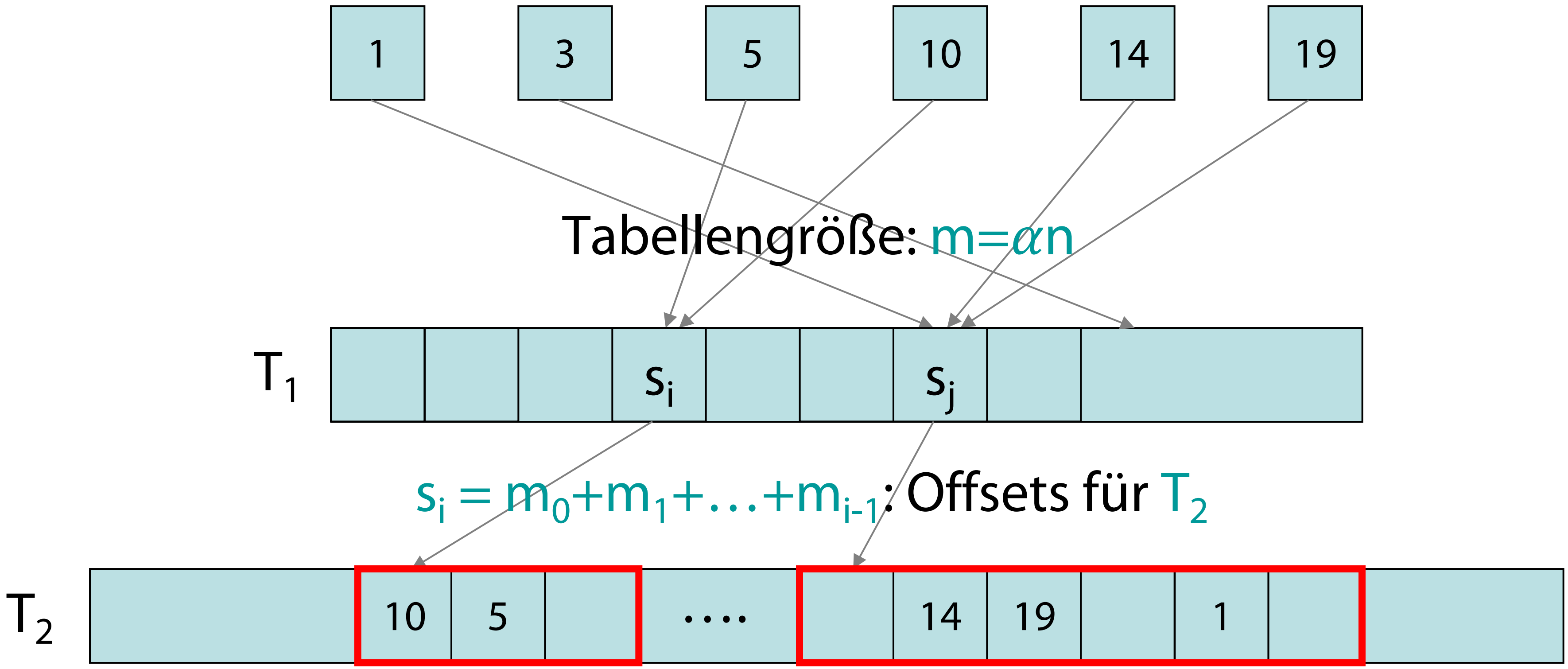
Statisches Wörterbuch (FKS-Hashing)



Wähle Subtabellengröße und Hashfunktion so, dass keine Kollisionen auftreten

Michael Fredman, János Komlós, Endre Szemerédi, Storing a Sparse Table with O(1) Worst Case Access Time, Journal of the ACM, Volume 31, Issue 3, **1984**

Statisches Wörterbuch



Statisches Wörterbuch

Behauptung: Für jede Menge von n Schlüsseln gibt es eine perfekte Hashfunktion der Größe $\Theta(n)$, die in erwarteter Zeit $\Theta(n)$ konstruiert werden kann.

Sind perfekte Hashfunktionen auch dynamisch konstruierbar??

Hashing: Prüfsummen und Verschlüsselung

- Bei Prüfsummen verwendet man Hashwerte, um Übertragungsfehler zu erkennen
  - Bei guter Hashfunktion sind Kollisionen selten,
  - Änderung weniger Bits einer Nachricht (Übertragungsfehler) sollte mögl. anderen Hashwert zur Folge haben
- In der Kryptologie werden spezielle kryptologische Hashfunktionen verwendet, bei denen zusätzlich gefordert wird, dass es praktisch unmöglich ist, Kollisionen absichtlich zu finden (→ SHAx, MD5)
  - Inverse Funktion $h^{-1}: T \longrightarrow U$ „schwer“ zu berechnen
  - Ausprobieren über $x=h(h^{-1}(x))$ ist „aufwendig“ da $|U|$ „groß“

Vermeidung schwieriger Eingaben

- Annahme: Pro Typ **nur eine Hash-Funktion** verwendet
- Wenn man Eingaben, die per Hashing verarbeitet werden, geschickt wählt, kann man **Kollisionen** durch geschickte Wahl der Eingaben **provozieren** (ohne gleiche Eingaben zu machen)
- Problem: Performanz sinkt (wird u.U. linear)
  - „Denial-of-Service"-Angriff möglich
- Lösung: Wähle Hashfunktion zufällig aus Menge von Hashfunktionen, die unabhängig von Schlüsseln sind

  → Universelles Hashing

Änderung der Hashfunktion bei Hashtabelle

- Hash-Funktion ändern beim Vergrößern oder Verkleinern einer Hashtabelle (Rehash)
- Messen der mittleren #Sondierungen und ggf. ein spontanes Rehash mit anderer Hash-Funktion
  - (latente Gefahr eines DOS-Angriffs abgemildert)
- Hierzu notwendig:
  - Auswahlmöglichkeit von h aus Menge von universell verwendbaren Hashfunktionen H

Universelles Hashing[1]

- Eine Menge von Hashfunktionen H heißt universell, wenn für beliebig wählbare Schlüssel x, y ∈ U mit x ≠ y gilt:
$$\frac{|\ \{h \in H \mid h(x) = h(y)\}\ |}{|H|} \leq 1/m$$
- Wenn eine Hashfunktion h aus H zufällig gewählt wird, ist die relative Häufigkeit von Kollisionen kleiner als 1/m, wobei m die Größe der Hashtabelle ist
- Beispiel: $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$
- Die Funktionen in H haben Parameter a, b
- Wir sprechen auch von einer Familie von Hashfunktionen

[1] Manchmal auch universales Hashing genannt: Für alle Schlüsselsequenzen geeignet, also universal einsetzbar

Universelles Hashing

Wir wählen eine Primzahl p, so dass jeder Schlüssel k kleiner als p ist.

$$Z_p = \{0,1,\ldots,p-1\}$$

$$Z_p^* = \{1,\ldots,p-1\}$$

Da das Universum erheblich größer als die Tabelle T sein soll, muss gelten:

$$p > m$$

Für jedes Paar (a,b) von Zahlen mit $a \in Z_p^*$ und $b \in Z_p$ definieren wir wie folgt eine Hash-Funktion:

$$h_{a,b}(k) = ((ak+b) \bmod p) \bmod m$$

Beispiel: p = 17 m = 6 ⇨ $h_{3,4}(8) = 5$

Universelles Hashing

Beh: Die Klasse $H_{p,m} = \{h_{a,b} \mid a \in Z_p^* \wedge b \in Z_p\}$

mit $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$

von Hash-Funktionen ist universell.

Ohne Beweis

Carter, Larry; Wegman, Mark N. "Universal Classes of Hash Functions". Journal of Computer and System Sciences. 18 (2): 143–154, **1979**.

Wörterbücher in Julia

julia> typeof(1=>"eins")
Pair{Int64, String}

- julia> Dict()
  ```
  Dict{Any, Any}()
  ```
- julia> d = Dict(1=>"eins", 5=>"fünf", 42=>"zw&vierzig")
  ```
  Dict{Int64, String} with 3 entries:
        5  => "fünf"
        42 => "zw&vierzig"
        1  => "eins"
  ```
- julia> d[1]
  ```
  "eins"
  ```
- julia> d[12]
  ```
  ERROR: KeyError: key 12 not found
  ```
- julia> d[12] = "zwölf"
  ```
  "zwölf"
  ```
- julia> d[12]
  ```
  "zwölf"
  ```

Zusammenfassung: Hashing

- Basisoperation (Suchen, Einfügen, Löschen) in O(1)
- Güte des Hashverfahrens beeinflusst durch
  - Hashfunktion
  - Verfahren zur Kollisionsbehandlung
    - Verkettung
    - Offene Adressierung
      - Lineares/Quadratisches Sondieren/Doppel-Hashing
  - Füllfaktor
    - Dynamisches Wachsen
- Statistisches vs. Dynamisches Hashen
- Universelles Hashing