Algorithmen und Datenstrukturen

Prof. Dr. Ralf Möller

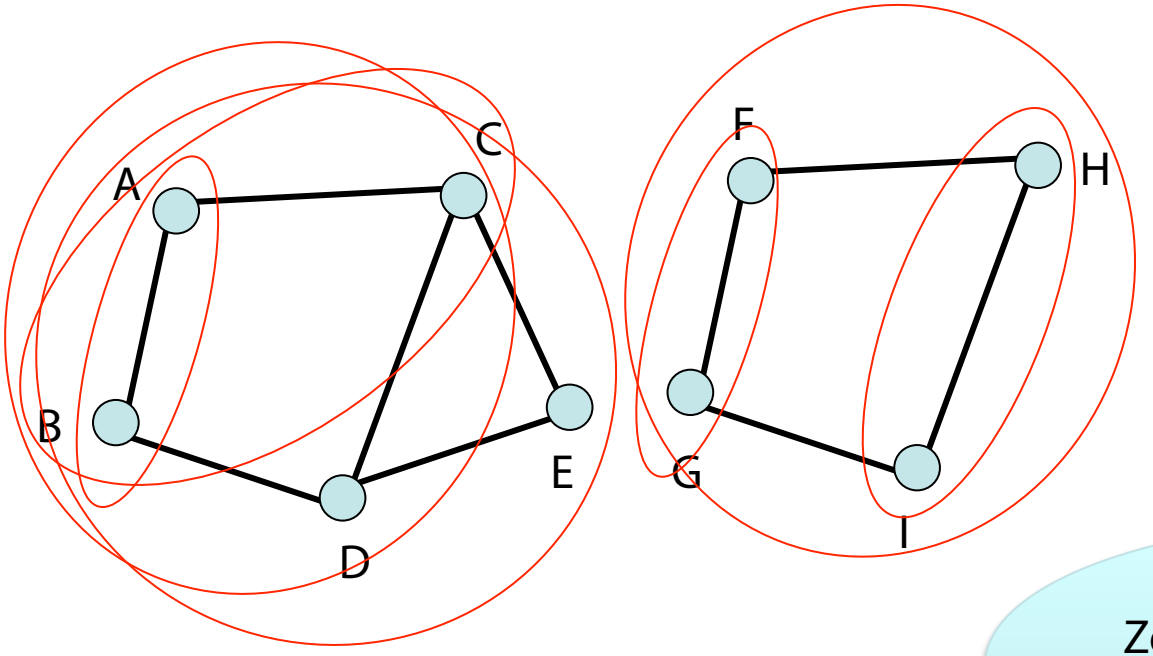
Universität zu Lübeck

Institut für Informationssysteme

Stefan Werner (Übungen)

sowie viele Tutoren

Wiederholung: Disjunkte Mengen (Union-Find)



procedure connected?(u, v, s):
  return find(u, s) = find(v, s)

- connected?("A", "E", s) ⟶ ?
- connected?("A", "H", s) ⟶ ?

Wir müssen von der Zeichenkette ausgehend den zugordneten Knoten in ~O(1) finden

Einsicht:
Keine aufwendige Suche nötig!

Danksagung

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

- „Effiziente Algorithmen und Datenstrukturen“ (Kapitel 2: Suchstrukturen) 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 Elementen

Jedes Element e identifiziert über key(e)

Operationen:

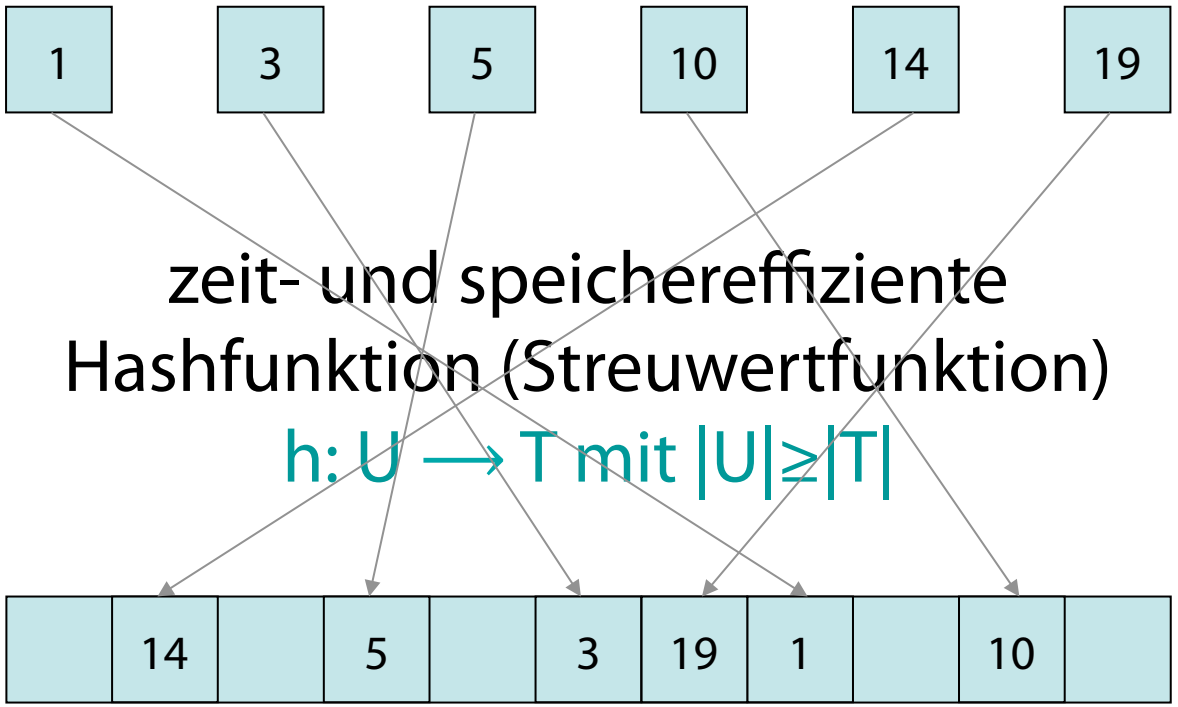
- insert(e, s): $s := s \cup \{e\}$ // Änderung nach außen sichtbar
- delete(k, s): $s := s \setminus \{e\}$, wobei e das Element ist mit key(e)=k // Änderung von s nach außen sichtbar
- lookup(k, s): Falls es ein $e \in S$ gibt mit key(e)=k, dann gib e aus, sonst gib $\perp$ aus

Wörterbücher

Unterschied zur Menge:

- Elemente (Einträge) bei Wörterbüchern als Attribut-Wert-Paare verstanden
- Iteration über Elemente eines Wörterbuchs in **willkürlicher** Reihenfolge **ohne** Angabe eines Bereichs
  - Über alle Attribute
  - Über alle Werte
  - Über alle Attribut-Wert-Paare

Hashing (Streuung)



zeit- und speichereffiziente
Hashfunktion (Streuwertfunktion)

$h: U \longrightarrow T$ mit $|U| \geq |T|$

Hashtabelle T (T = Indexbereich)

Motivation für den Namen Hashing

Zerhacker früher im Sprechfunk eingesetzt (vor der Digitaltechnik)

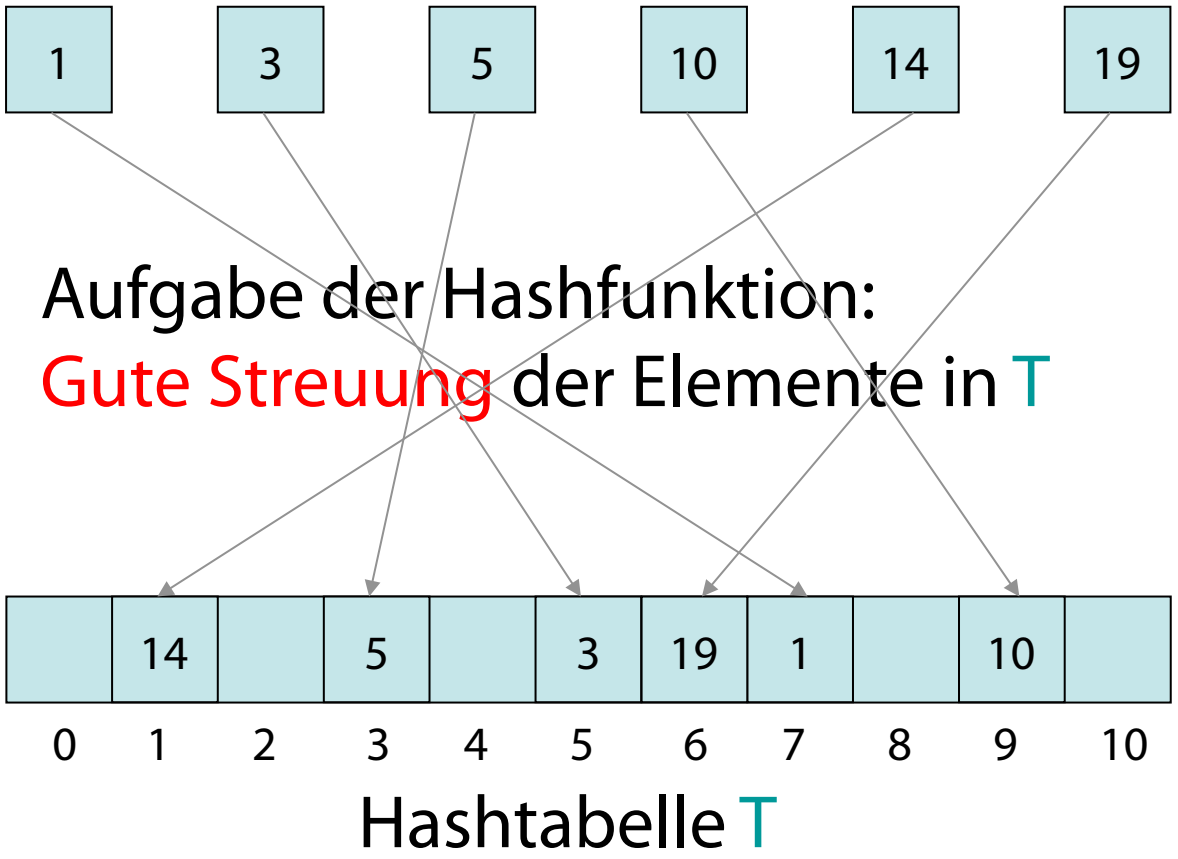
Zerhacken der Sprache in kleine Segmente, die mit einem Ringmodulator in verschiedene Frequenzbereiche verteilt (oder: gestreut) wurden

Sprache für einen Zuhörer unverständlich

Hashing: Übliches Anwendungsszenario

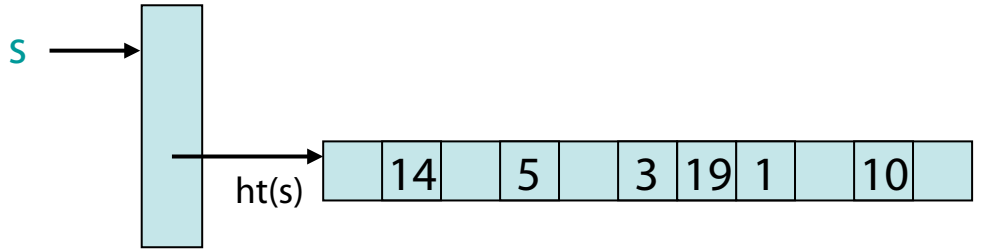
- Menge U der potentiellen Schlüssel u „groß“
- Anzahl der Feldelemente length(T) „klein“
- D.h.: |U| >> length(T), aber nur „wenige“ u ∈ U werden tatsächlich betrachtet
- Werte u können „groß“ sein (viele Bits)
  - Große Zahlen, Tupel mit vielen Komponenten, Bäume, ...
- Werte u können „zerhackt“ werden, so dass evtl. nur Teile von u zur einfachen Bestimmung des Index für T betrachtet werden
  - Z.B.: Nur einige Komponenten eines Tupel betrachtet
  - Bäume nur bis zu best. Tiefe betrachtet
- Sonst Abbildungsvorgang evtl. zu aufwendig

Hashing


1
3
5
10
14
19
Aufgabe der Hashfunktion:
Gute Streuung der Elemente in T
14
5
3
19
1
10
0 1 2 3 4 5 6 7 8 9 10
Hashtabelle T

Hashing (perfekte Streuung)

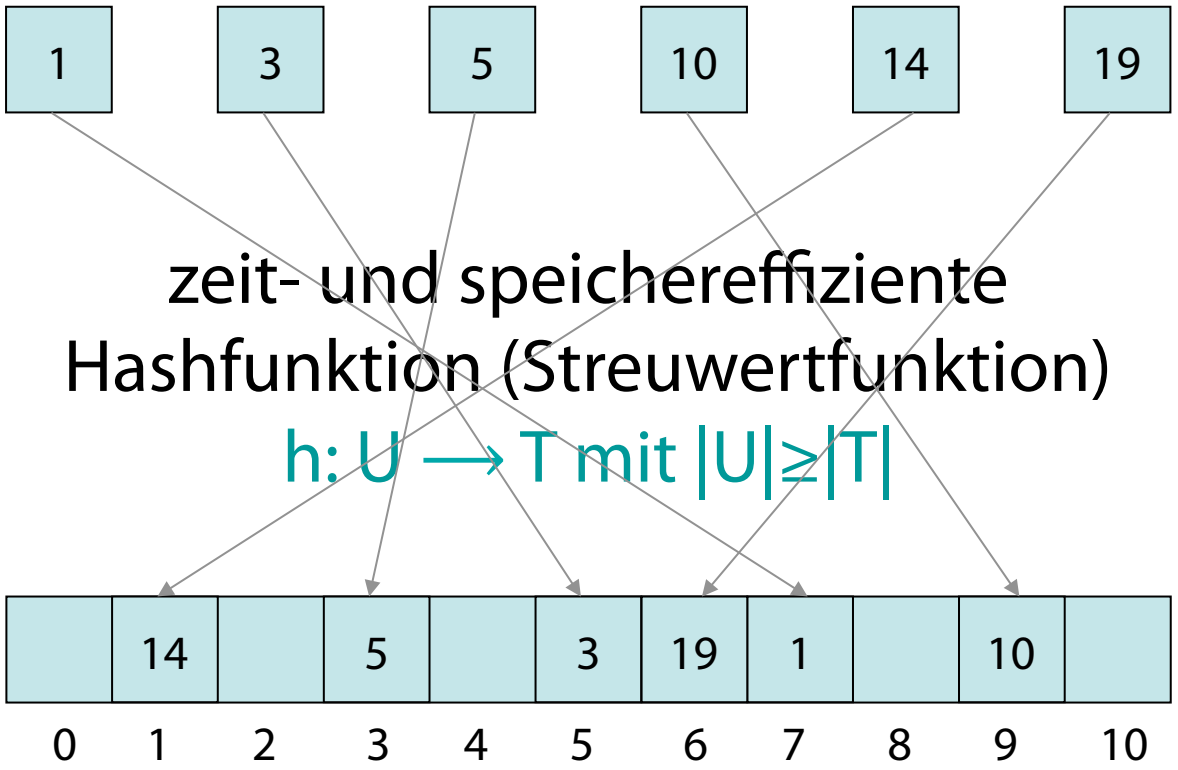
```
procedure insert(e, s):
    T := ht(s)
    T[h(key(e))] := e
```




```
procedure delete(k, s):
    T := ht(s)
    if key(T[h(k)])=k then T[h(k)] := ⊥

function lookup(k, s):
    T := ht(s)
    return T[h(k)]
```

Hashing (Streuung)

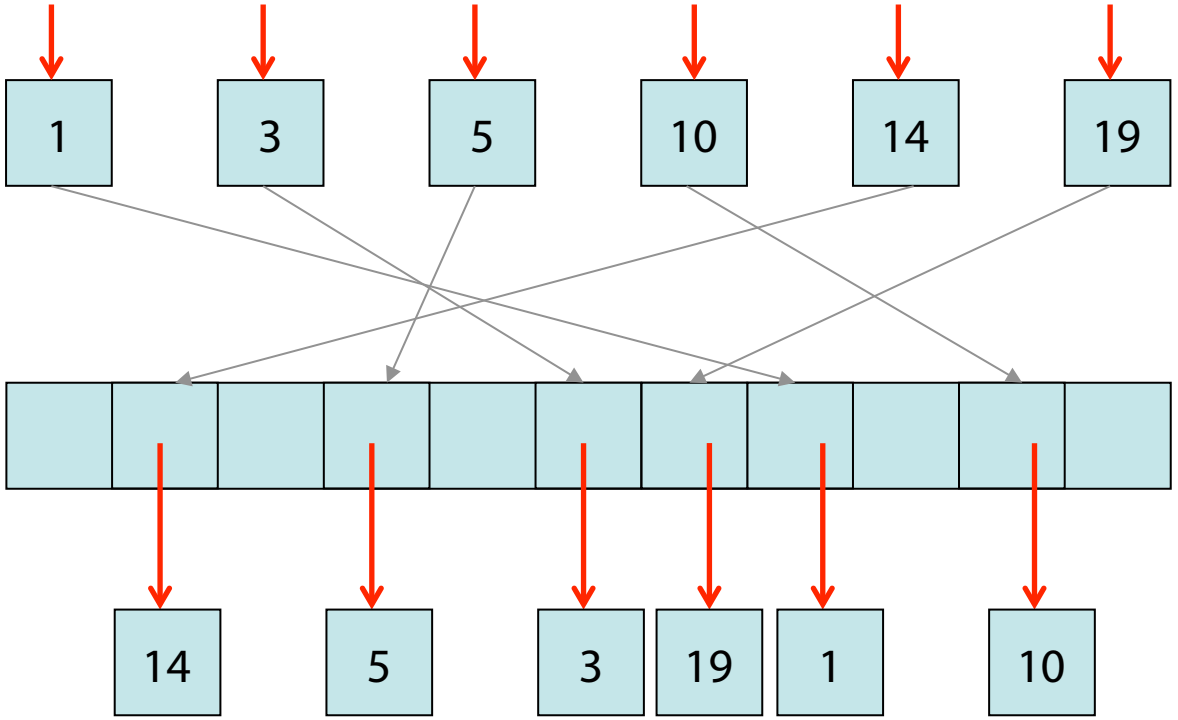
Einige Elemente
aus einer Menge U:



Assoziation durch Hashing sowie Iteration

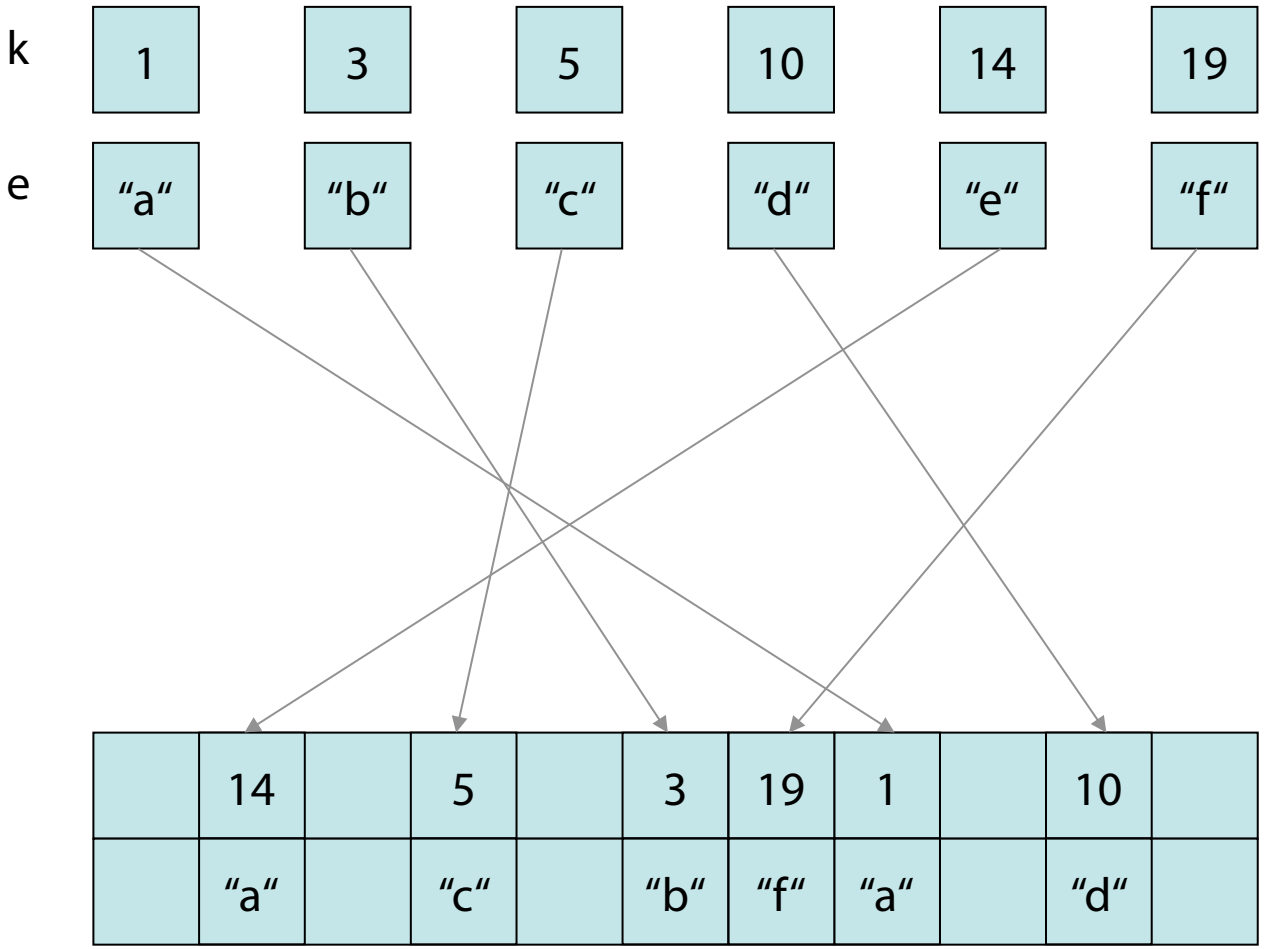
- Elemente von U sind nicht notwendigerweise „nur“ Zahlen
- Anwendung eines Schlüsselfunktion key: k=key(e)

Zahlen in der Box stellen Schlüsselwerte key(e) dar

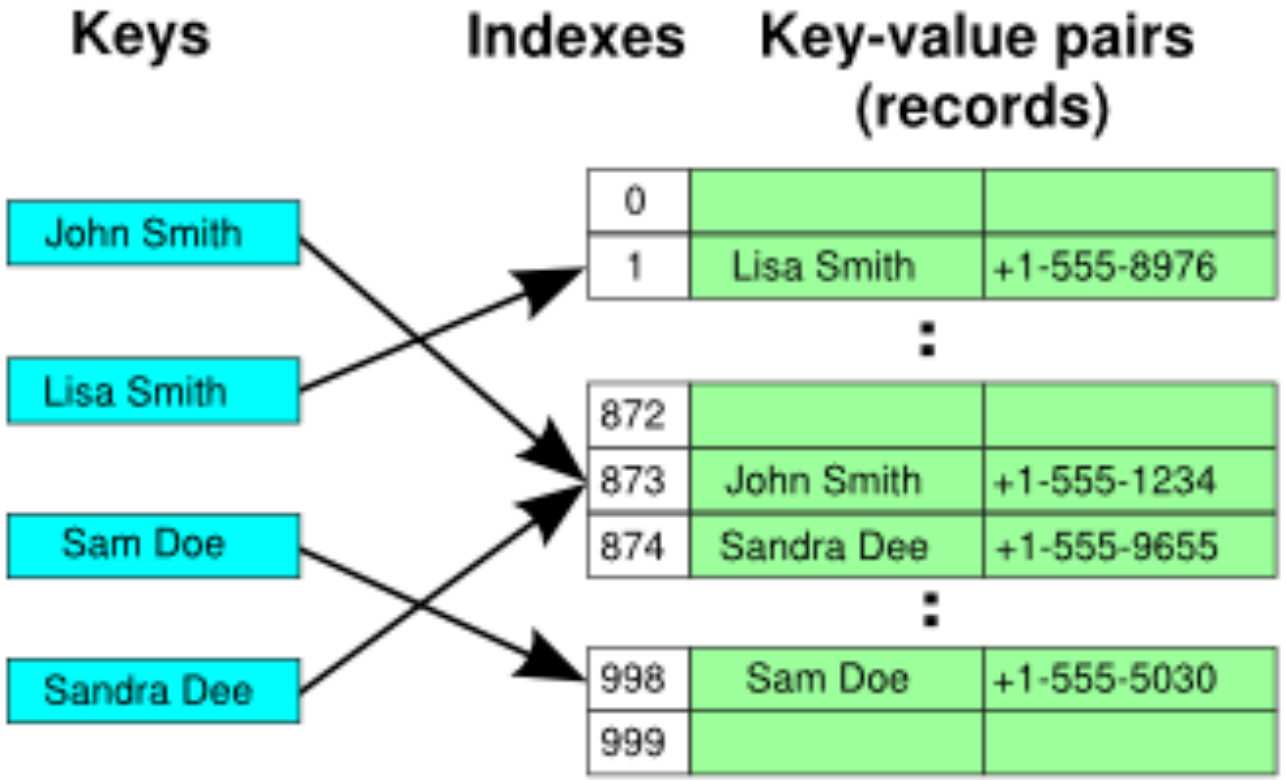


- Schlüssel selbst können auch Objekte sein

insert(e, s) vs. insert(k, e, s)



Praktische Anwendung von insert(k, e, s)



Hashfunktionen

- Hashfunktionen müssen i.A. anwendungsspezifisch definiert werden (oft für Basisdatentypen Standardimplementierungen angeboten)
- Hashwerte sollen möglichst gleichmäßg 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 sei 100
→ Viele Kollisionen

Warum i.A. nicht?

- Kann man hash auf komplexe Objekte über deren „Adresse“ durchführen?

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

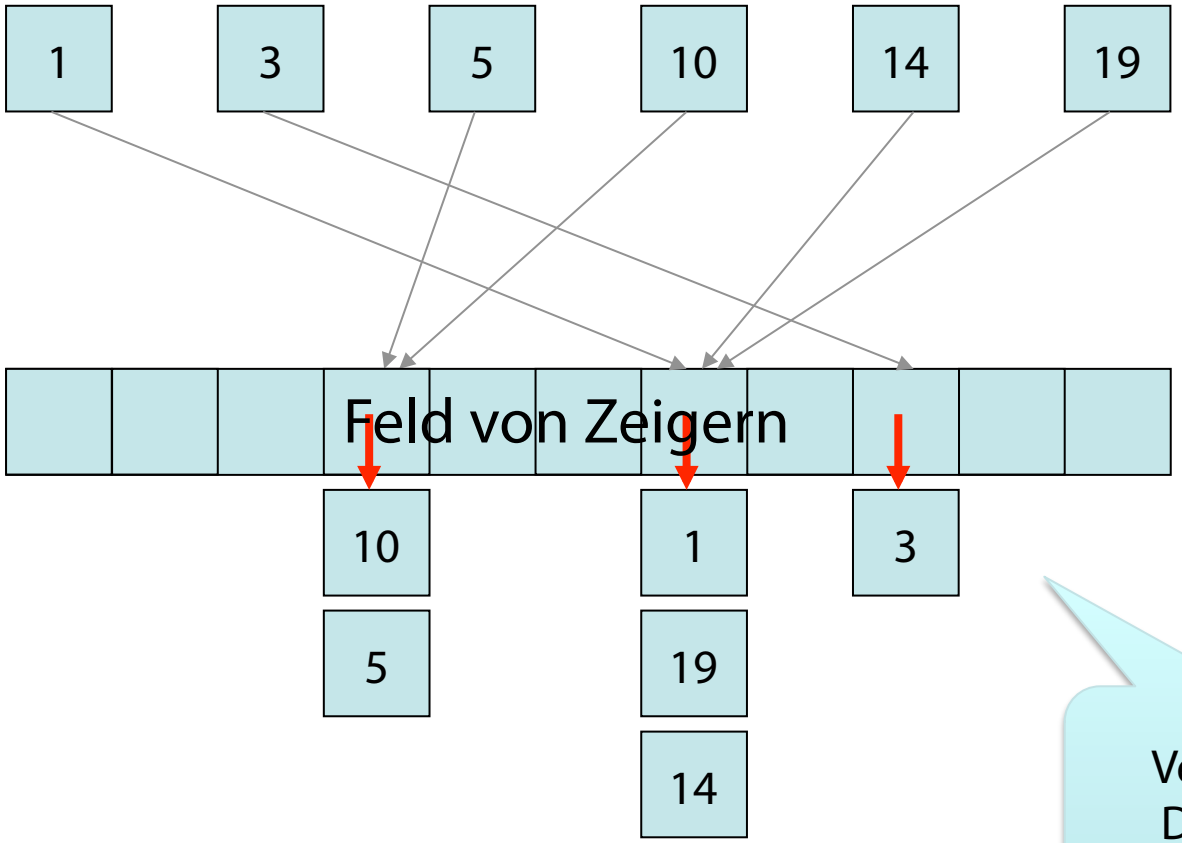
Hashfunktionen

- Beim Erzeugen einer Hashtabelle kann man die initiale Größe angeben
- Man möchte den Nutzer nicht zwingen, eine Primzahl zu verwenden
- Andere Hashfunktion für U = Integer:

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

wobei p>m eine „interne“ Primzahl und m=length(T) nicht notwendigerweise prim

Hashing mit Verkettung[1] (Kollisionslisten)



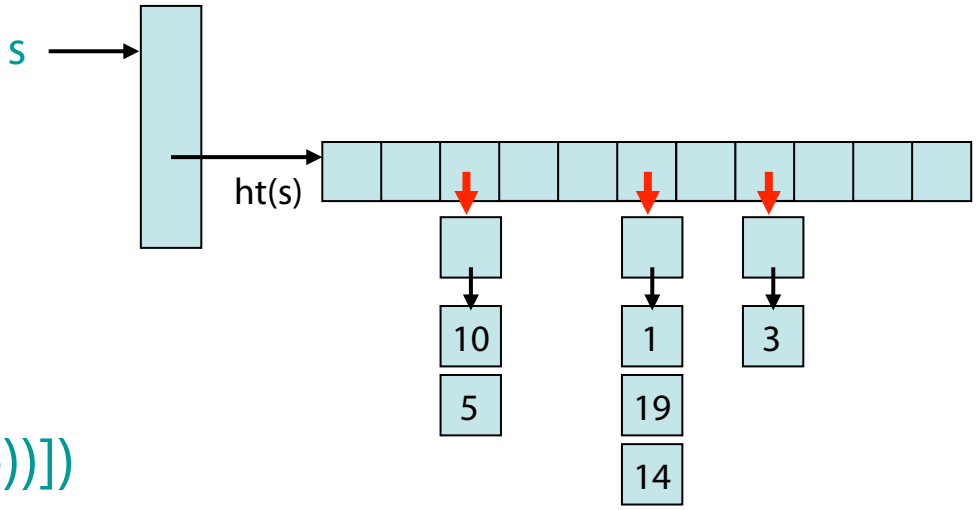
unsortierte verkettete Listen

Vereinfachte Darstellung

[1] Auch geschlossene Adressierung genannt.

Hashing mit Verkettung

- T: Array [0..m-1] of List
- List: Datenstruktur mit Operationen:
  insert, delete, lookup

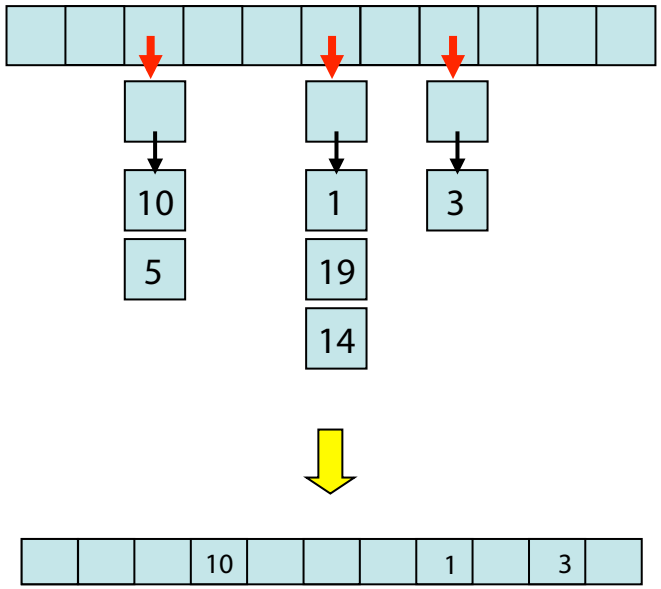



```
procedure insert(e, s):
    T := ht(s); insert(e, T[h(key(e))])

procedure delete(k, s):
    T := ht(s); delete(k, T[h(k)])

function lookup(k, s):
    T := ht(s)
    return T[h(k)].lookup(k)
```

Analyse der Komplexität bei Verkettung

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

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



Dynamisches Wörterbuch

Problem: Hashtabelle kann zu groß oder zu klein sein
(sollte nur um konstanten Faktor abweichen von der Anzahl der Elemente)

Lösung: Reallokation

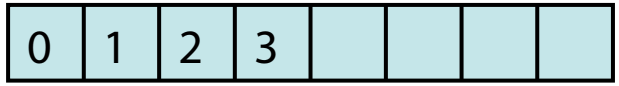
- Wähle neue geeignete Tabellengröße
- Wähle neue Hashfunktion
- Übertrage Elemente auf die neue Tabelle
  - Jeweils mit Anwendung der (neuen) Hashfunktion
  - In den folgenden Darstellung ist dieses nicht gezeigt!

Dynamische Hashtabelle

Vereinfachung!
(evtl. für Hashtabelle schon ab n>m/2 nötig)

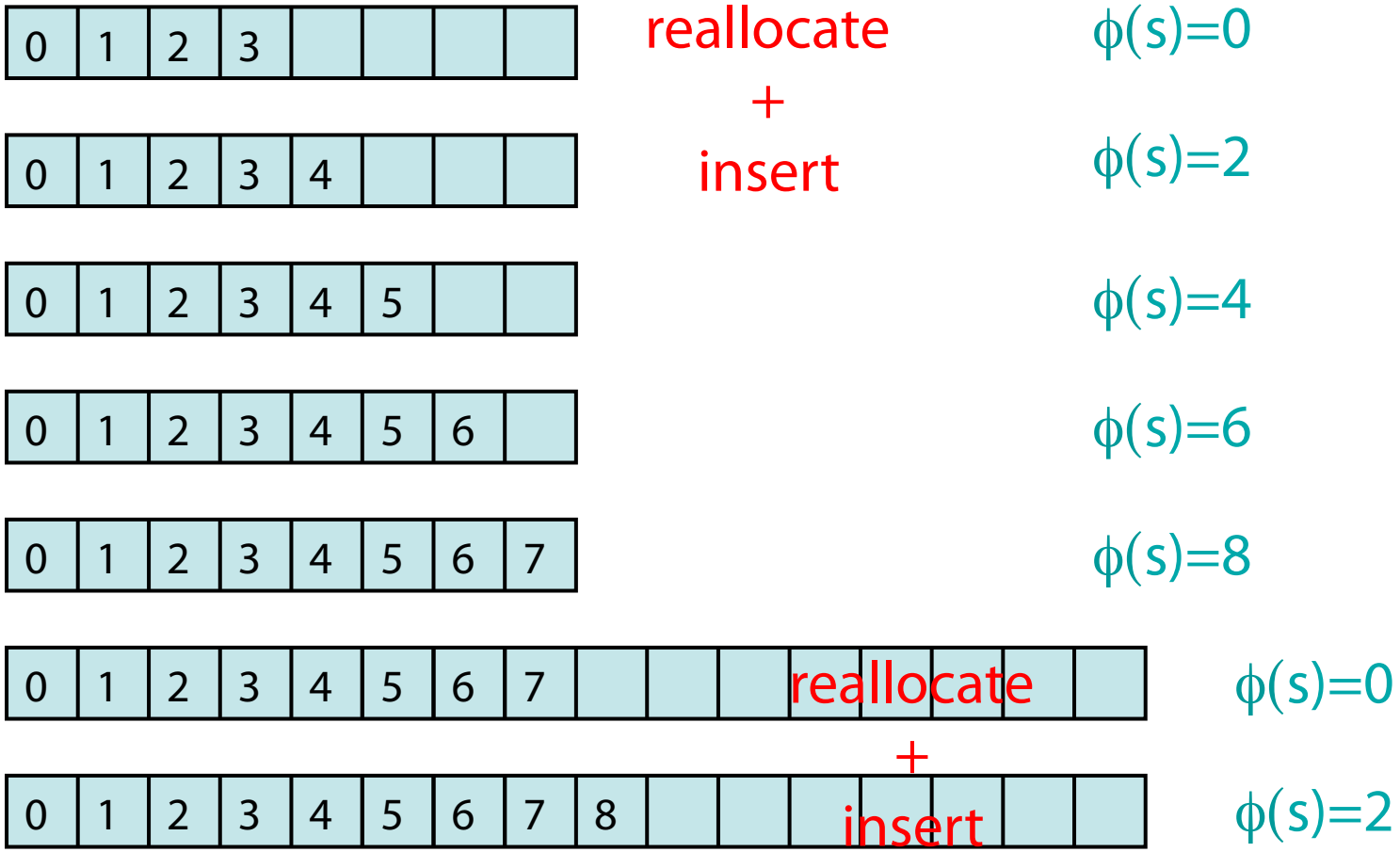
- Tabellenverdopplung (n>m):

- Tabellenhalbierung (n≤m/4):

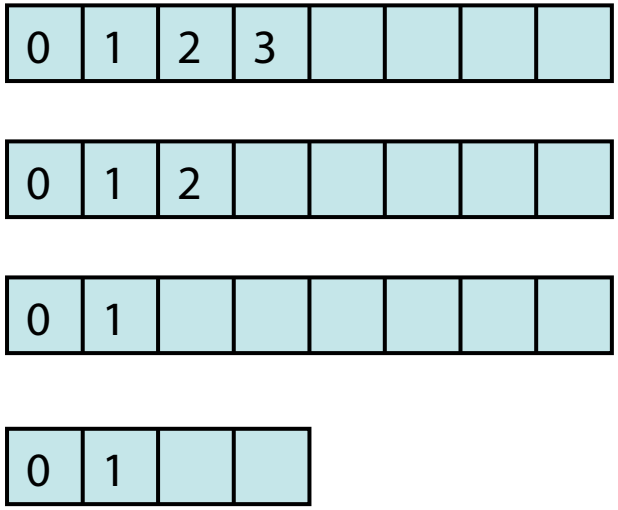
- Von 

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

Dynamische Hashtabelle



Dynamische Hashtabelle



$\phi(s)=0$

$\phi(s)=2$

$\phi(s)=4$

$\phi(s)=0$

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)

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

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

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

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
  - Tabelle muss größer sein (maximaler Füllfaktor kleiner) als bei Verkettung, um konstante Komplexität bei den Basisoperationen zu erreichen

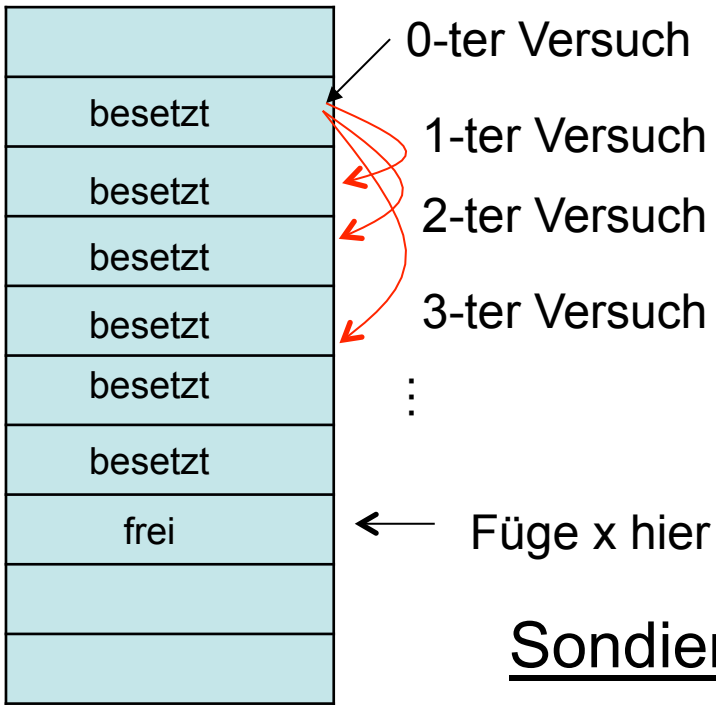
Offene Adressierung

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

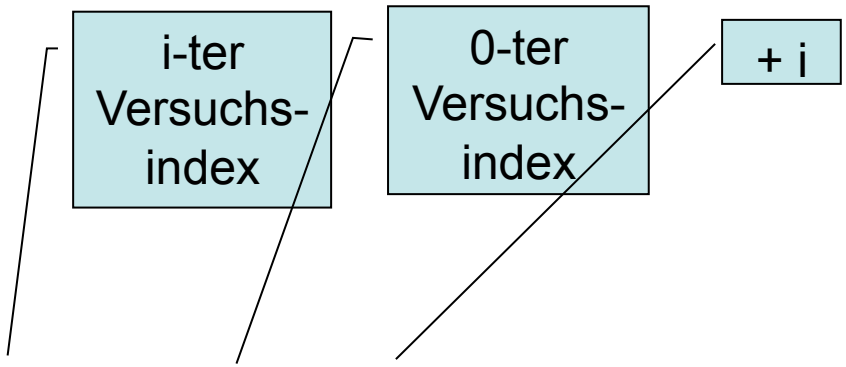
- Hashfunktion
  - $h_i(x) = (h(x) + f(i)) \bmod m$
  - $f(0) = 0$ Position des 0-ten Versuches
  - $f(i)$ „Distanz des i-ten Versuches relativ zum 0-ten Versuch“

Lineares Sondieren

Linear probing:



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



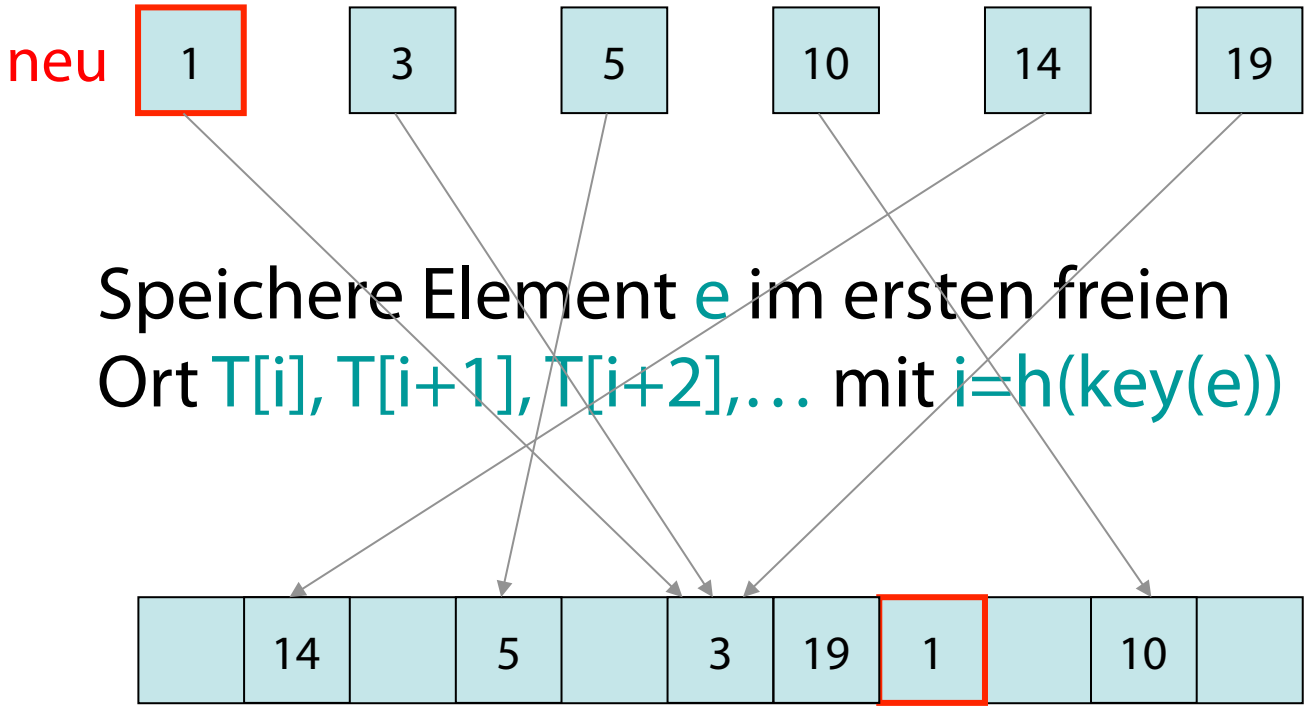
$h_i(x) = (h(x) + i) \bmod m$

Sondierungssequenz: +0, +1, +2, +3, +4, …

Fahre fort bis ein freier Platz gefunden ist

#fehlgeschlagene Versuche als eine Messgröße der Performanz

Hashing with Linearer Sondierung (Linear Probing)



Hashing with Linearer Sondierung

```
T: Array [0..m-1] of Element    // m>n

procedure insert(e, s)
    T := ht(s); i := h(key(e))
    while T[i]<> ⊥ & key(T[i])<>key(e) do
        i := (i+1) mod m
    T[i] := e

function lookup(k, s):
    i:=h(k); T:=ht(s)
    while T[i]<> ⊥ & key(T[i])<>k do
        i:=(i+1) mod m
    return T[i]
```

Hashing with Linearer Sondierung

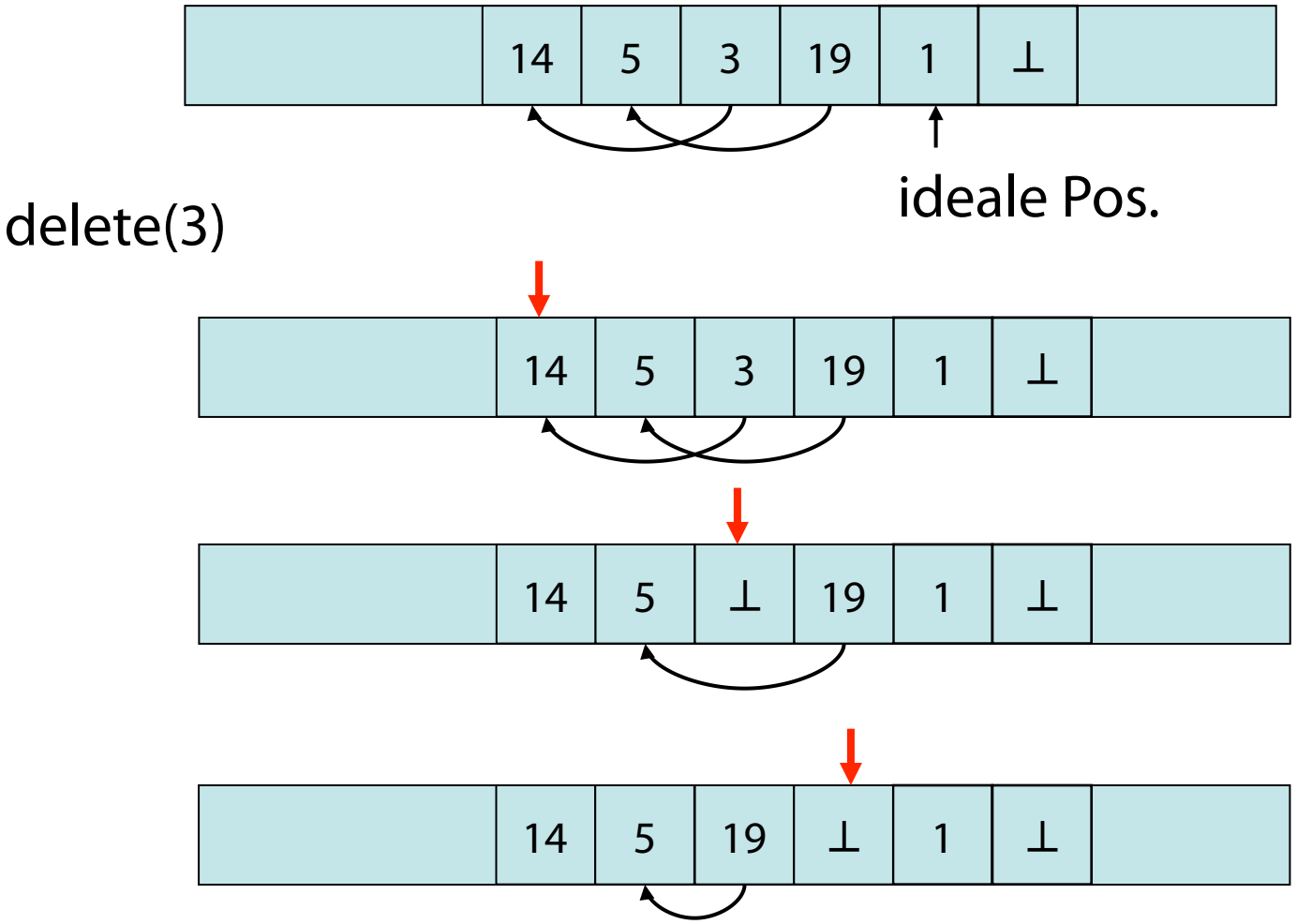
Problem: Löschen von Elementen

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 $i=h(key(e))$ und aktueller Position j gilt

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

Beispiel für Lösch-Operation



Hashing with Linearer Sondierung

```
Procedure delete(k: Key)
    i := h(k)
    j := i
    empty := false        // Indicator für T[i]=⊥
    while T[j]<>⊥ do
      if h(key(T[j]))<=i & empty then
          T[i] := T[j]
          T[j] := ⊥
          i := j          // repariere Invariante ab T[i]
      if key(T[j])=k then
          T[j] := ⊥       // entferne Element mit Schlüssel k
          i := j
          empty := true
      j := (j+1) mod m
```

Klappt nur, solange i>h(k). Wie sieht allg. Fall aus?

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, werden an das Ende des Clusters gespeichert (und 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

Beweis: An der Tafel

Beweis [AuD M. Hofmann LMU 06]

Sei X eine Zufallsvariable mit Werten aus $\mathbb{N}$.

Dann ist

$$E[X] := \sum_{i=0}^{\infty} i\Pr\{X = i\} = \sum_{i=1}^{\infty} \Pr\{X \geq i\}$$

Dies deshalb, weil $\Pr\{X \geq i\} = \sum_{j=i}^{\infty} \Pr\{X = j\}$.

Daher ergibt sich für die erwartete Suchdauer D:

$$D = \sum_{i=1}^{\infty} \Pr\{\text{„Mehr als } i \text{ Versuche finden statt“}\}$$

$$\Pr\{\text{„Mehr als } i \text{ Versuche finden statt“}\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+1}{m-i+1} \leq \alpha^i$$

Also, $D \leq \sum_{i=1}^{\infty} \alpha^i = 1/(1-\alpha)$.

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

Beweis: An der Tafel

Beweis aus [CLRS]

Proof A search for a key k reproduces the same probe sequence as when the element with key k was inserted. By Corollary 11.7, if k was the $(i+1)$st key inserted into the hash table, the expected number of probes made in a search for k is at most $1/(1-i/m) = m/(m-i)$. Averaging over all n keys in the hash table gives us the expected number of probes in a successful search:

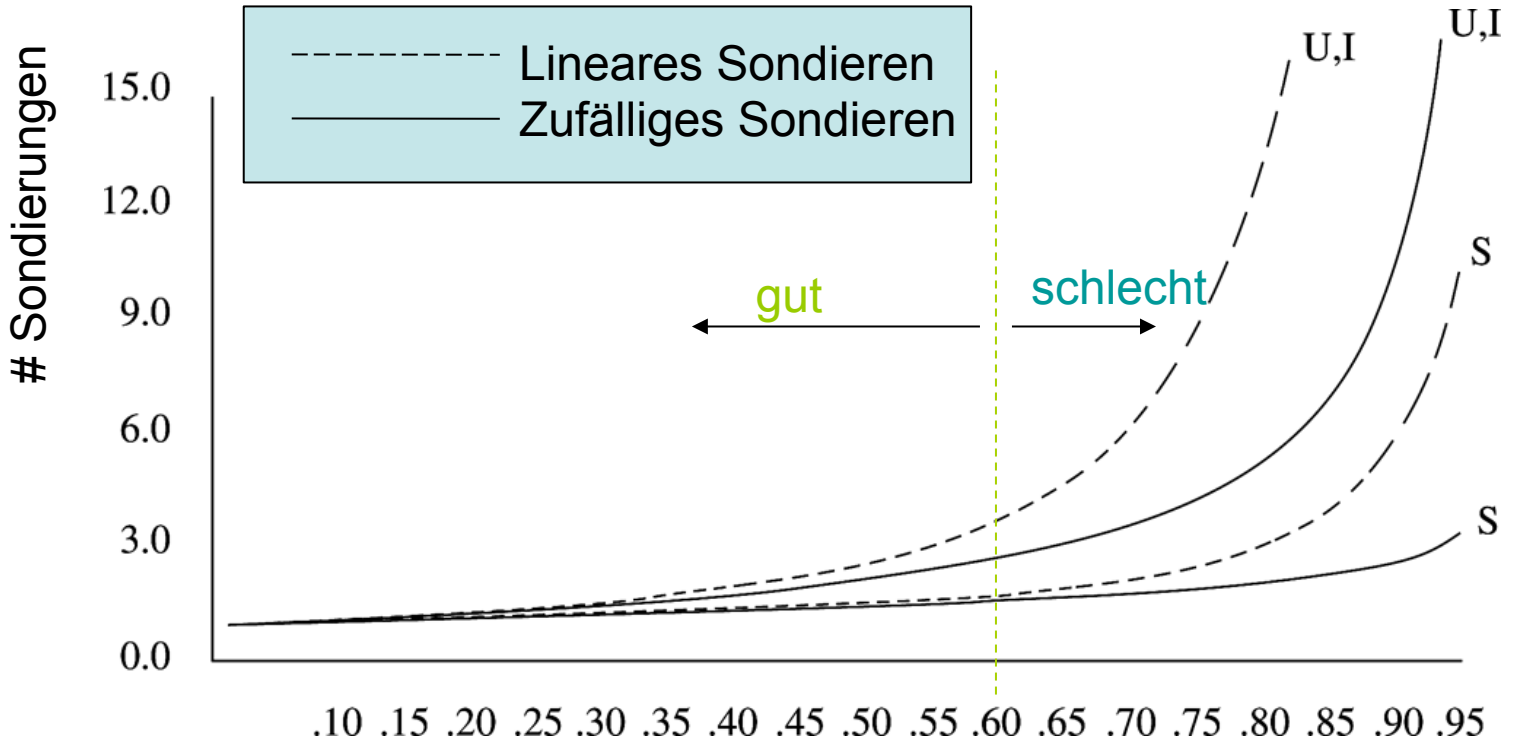
$$
\begin{aligned}
\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\
&= \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \\
&\le \frac{1}{\alpha}\int_{m-n}^{m}(1/x)\,dx \quad \text{(by inequality (A.12))} \\
&= \frac{1}{\alpha}\ln\frac{m}{m-n} \\
&= \frac{1}{\alpha}\ln\frac{1}{1-\alpha}\,.
\end{aligned}
$$

■

Zufälliges Sondieren

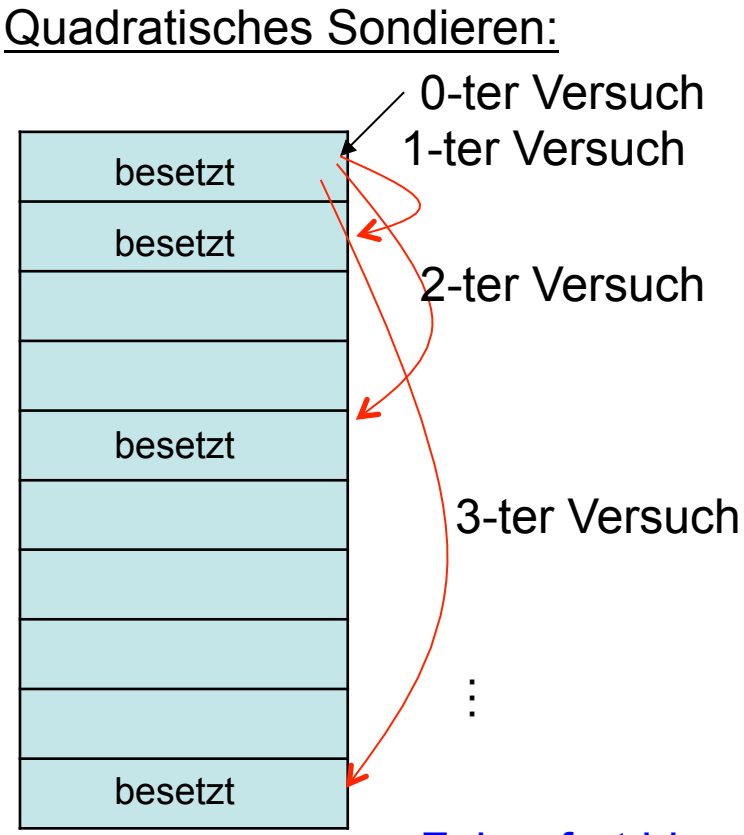
- Wähle den jeweils nächsten Feldindex nach einer (reproduzierbaren) Zufallsfolge
  - Sehr rechenaufwendig
- Für jeden Schlüssel k wähle genügend lange zufällige Versatzfolge f(i) und speichere Folge f(i) zur Verwendung bei erneutem Hash von k
  - Sehr speicheraufwendig
  - Man hat auch ein Bootstrap-Problem, da die Assoziation Key→Indexfolge mittels Hashing erfolgen wird

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, …
  - 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 $\alpha_{max} \leq ½$ für quadratisches Sondieren

Doppel-Hashing

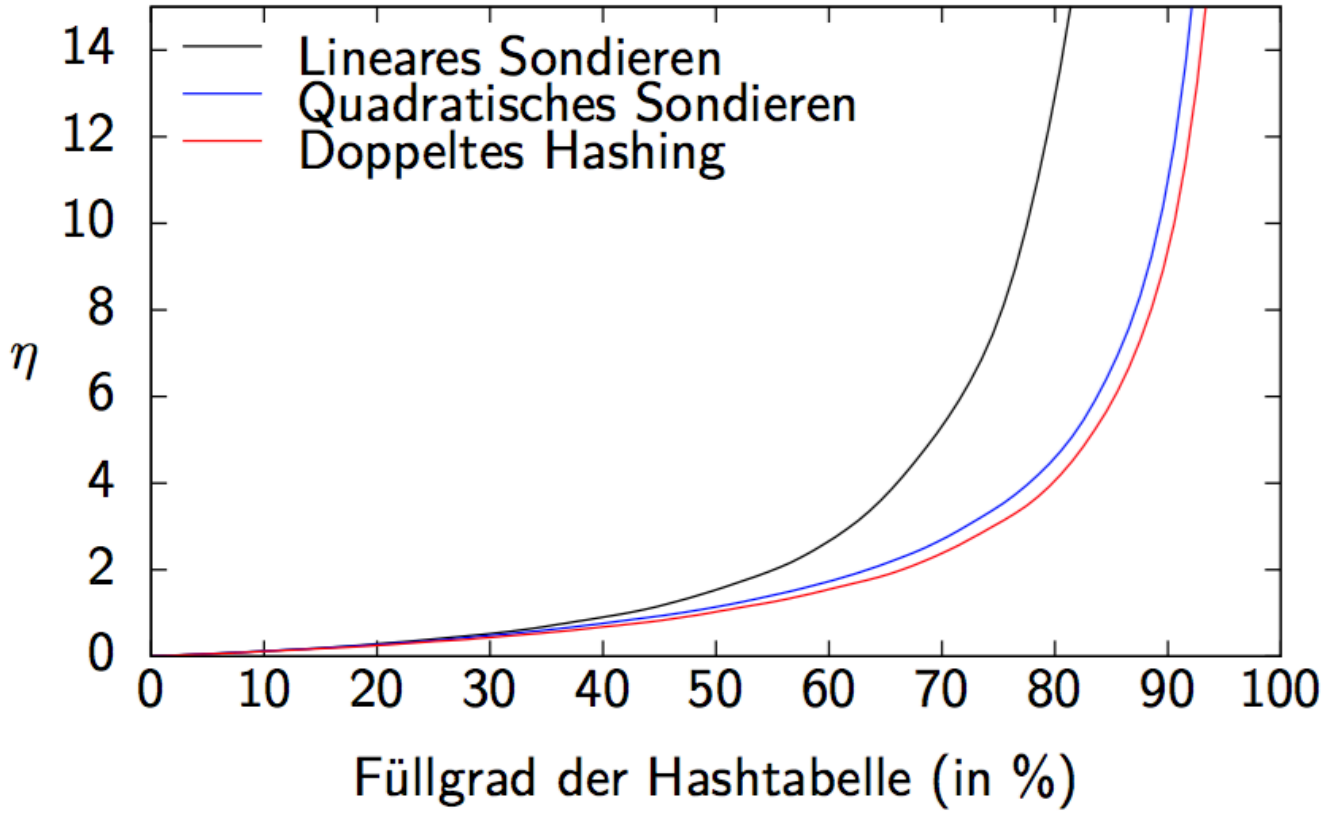
- Benutze eine zweite Hashfunktion für alle Versuche außer dem ersten $f(i) = i \cdot h'(x)$

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

- Beispiel für $m=10$.
  - $h_0(49) = (h(49) + f(0)) \bmod 10 = 9$
  - $h_1(49) = (h(49) + \underbrace{1 \cdot (7 - 49 \bmod 7)}_{f(1)}) \bmod 10 = 6$ für $p=7$

Praktische Effizienz von doppeltem Hashing

- Hashtabelle mit 538 051 Einträgen (Endfüllgrad 99,95%) 99,8 % -> 358
- *Mittlere* Anzahl Kollisionen η pro Einfügen in die Hashtabelle:

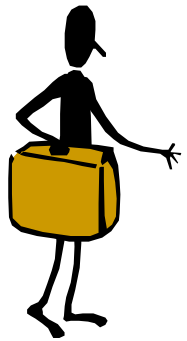


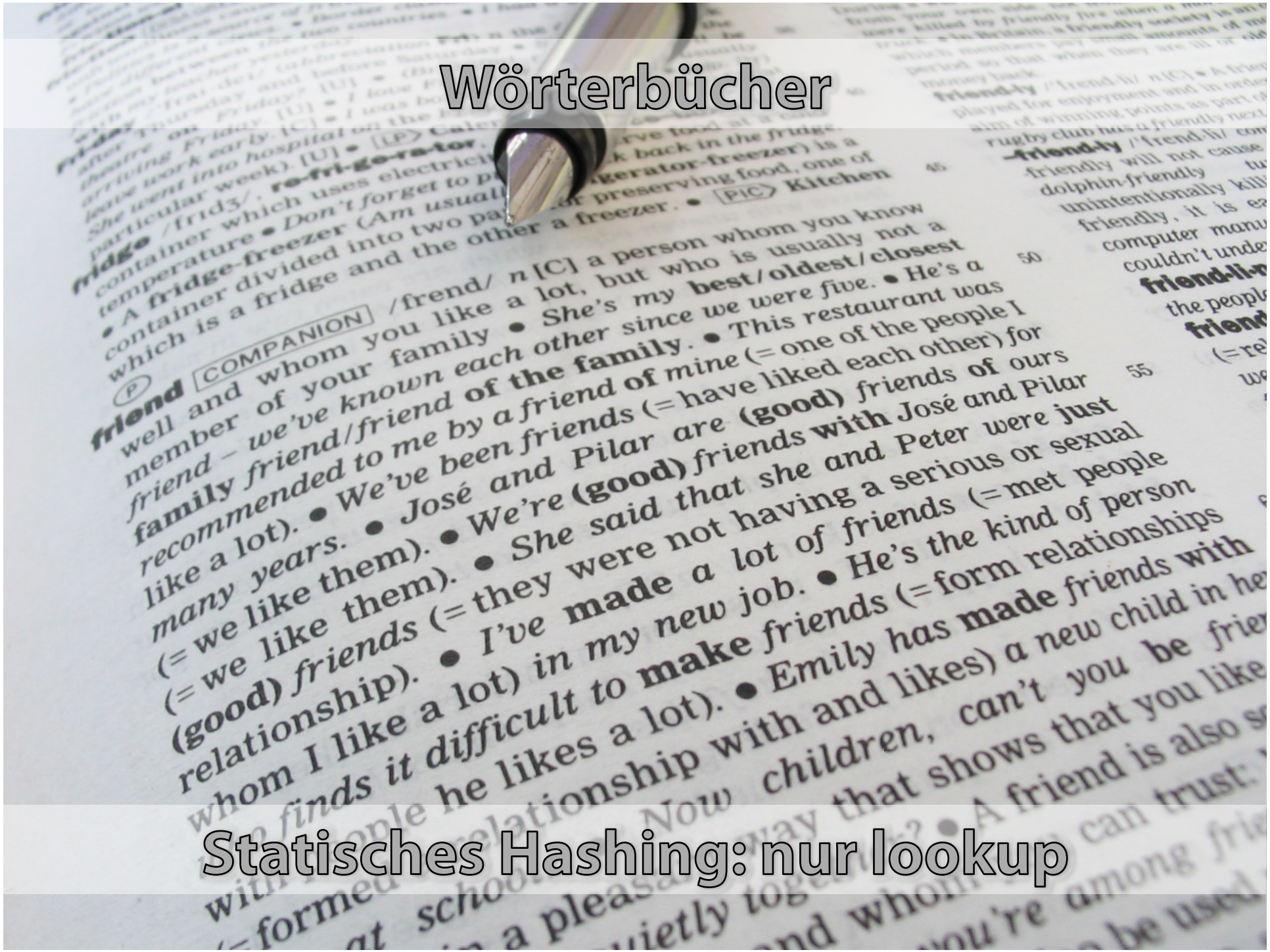
Analyse Hashing

- Messreihen zeigen, dass Doppel-Hashing fast genauso gut ist wie zufälliges Sondieren
- Zweite Hashfunktion benötigt zusätzliche Zeit zum Berechnen
- Doppeltes Hashing ist langsamer als Überlaufketten und lineares Sondieren bei dünn besetzten Tabellen, jedoch wesentlich schneller als lineares Sondieren, wenn der Füllgrad der Tabelle zunimmt
- Generell: Hashing ist bedeutend schneller ist in Bäumen
  - Aber: Keine Bereichsanfragen

Zusammenfassung: Hashing

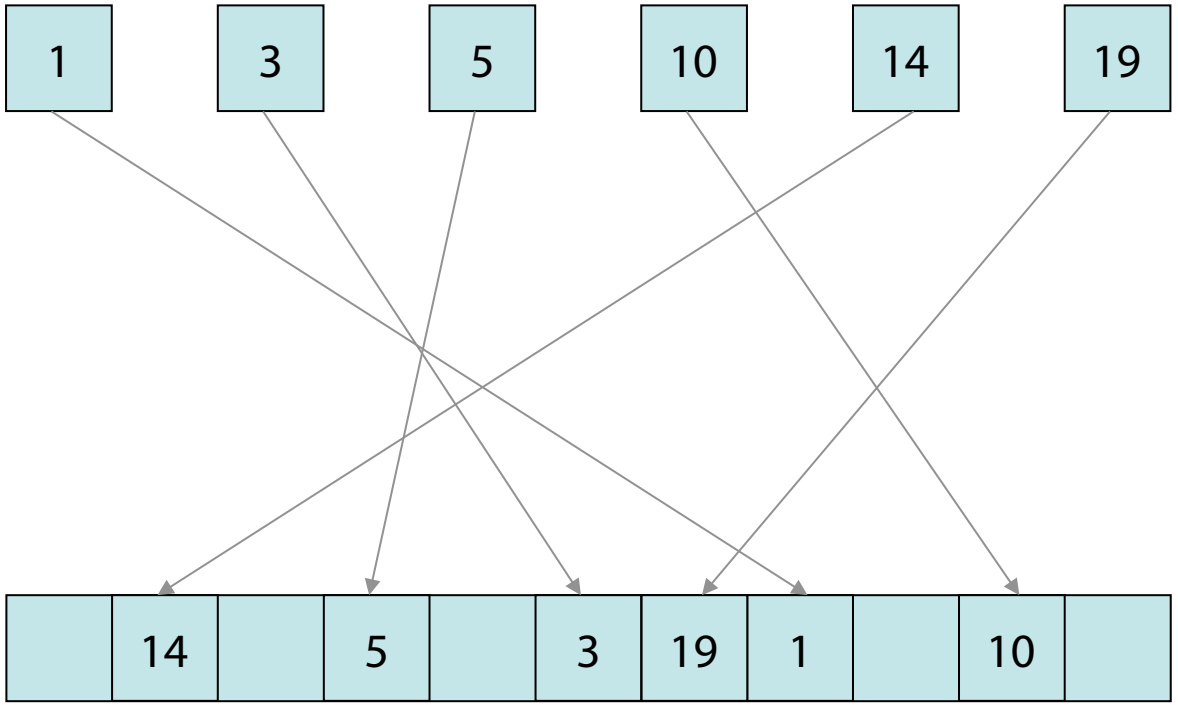
- Basisoperation (Suchen, Einfügen, Löschen) in O(1)
- Güte des Hashverfahrens beeinflusst durch
  - Hashfunktion
  - Verfahren zur Kollisionsbehandlung
    - Verkettung
    - Offene Adressierung
      - Lineares/Quadratisches Sondieren, Doppel-Hashing
  - Füllfaktor
    - Dynamisches Wachsen



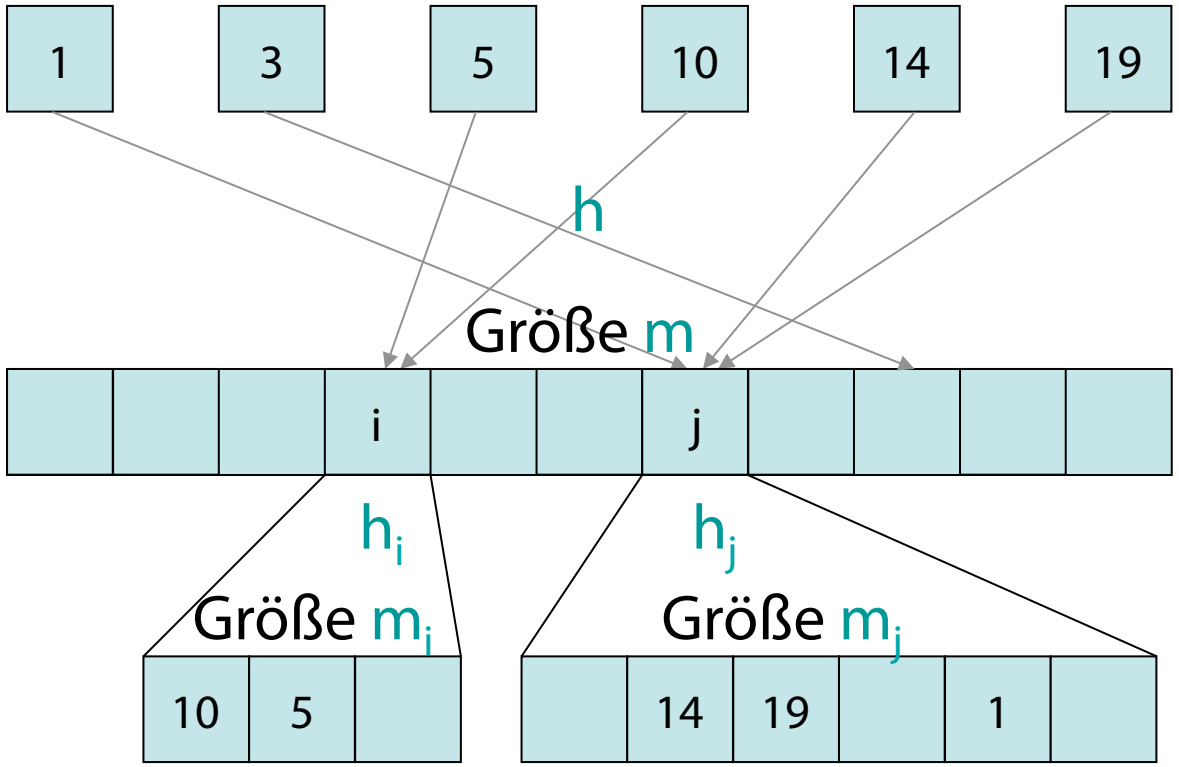

Wörterbücher
Statisches Hashing: nur lookup

Statisches Wörterbuch

Ziel: perfekte Hashtabelle



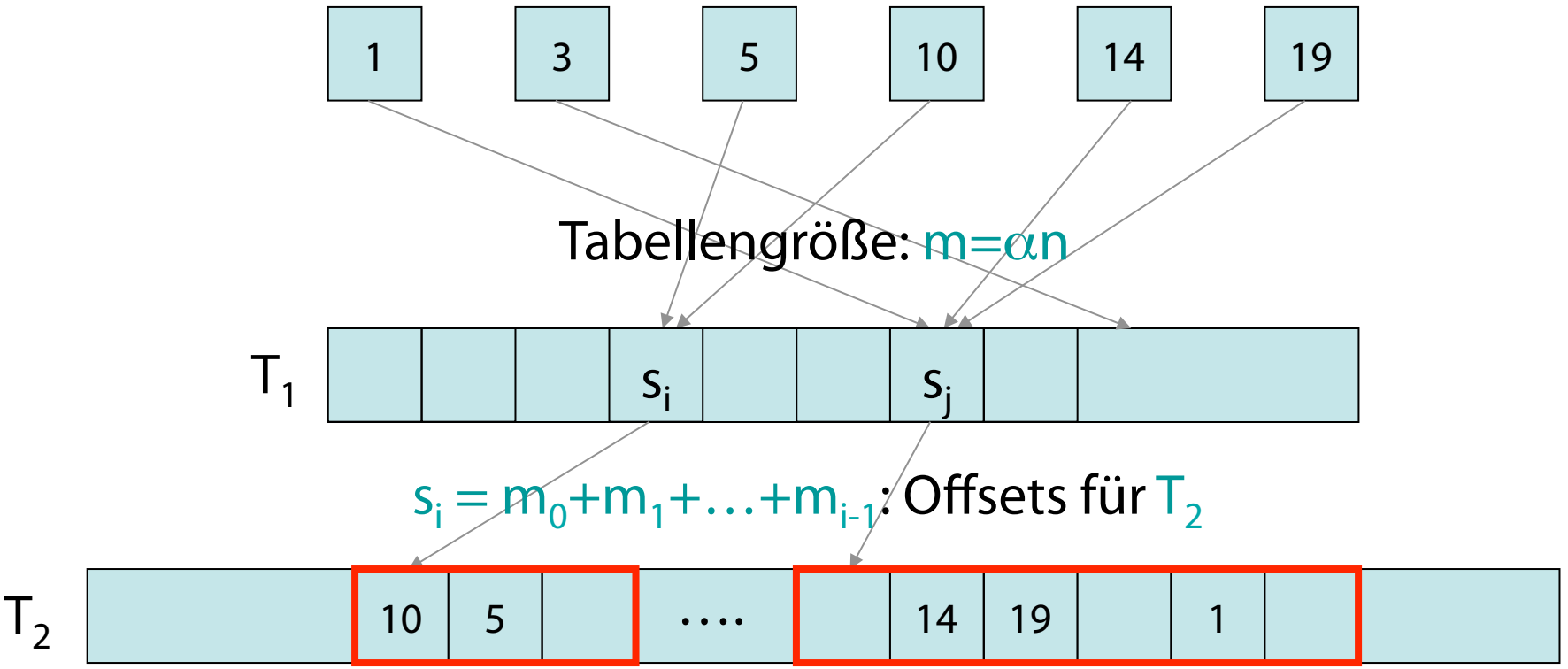
Statisches Wörterbuch (FKS-Hashing)



Keine Kollisionen in Subtabellen

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

Theorem: 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??

Perfektes Hashing zur Identifikation

=

```
79054025
255fb1a2
6e4bc422
aef54eb4
```

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

Universelles Hashing: Verwendung

- Auswahl der Hash-Funktion möglich, wenn Hashing für einen temporär aufgebauten Index verwendet wird,
  - Auswahl aus Menge von Hashfunktionen H möglich
- Bei langlaufenden Serveranwendungen kann Hash-Funktion beim Vergrößern oder Verkleinern einer Hashtabelle geändert werden (Rehash)
- Weiterhin kann die Zugriffszeit gemessen werden und ggf. ein spontanes Rehash mit einer anderen Hash-Funktion eingeleitet werden (also latente Gefahr eines DOS-Angriffs abgemildert)

Universelles Hashing[1]

- Eine Menge von Hashfunktionen H heißt universell, wenn für beliebige Schlüssel x, y ∈ U mit x ≠ y gilt:

$$\frac{|\{h \in H \mid h(x) = h(y)\}|}{|H|} \leq 1/m$$

- Also: 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

[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

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

von Hash-Funktionen ist universell.

Beweis:

Wir betrachten zwei beliebige, verschiedene Schlüssel k_i und k_j aus Z_p und eine gegebene Hash-Funktion $h_{a,b} \in H_{p,m}$:

$$r = (ak_i + b) \bmod p \qquad s = (ak_j + b) \bmod p$$

Zunächst stellen wir fest, dass r ≠ s ist, da

$$r - s \equiv (a(k_i - k_j)) \bmod p$$

$a \neq 0$, $(k_i - k_j) \neq 0$ und p eine Primzahl ist.

Es gibt also modulo p keine Kollisionen.

Universelles Hashing

Darüber hinaus liefert jedes der möglichen p(p-1) Paare (a,b) mit a ≠ 0 ein anderes Paar (r,s) von Resultaten modulo p:

$$a = (r - s)((k_i - k_j)^{-1} \bmod p) \bmod p$$

$$b = (r - ak_i) \bmod p$$

Es gibt also eine Eins-zu-eins-Beziehung (Bijektion) zwischen den p(p-1) Paaren (a,b) mit a ≠ 0 und den Paaren (r,s) mit r ≠ s.

Wenn wir also für ein beliebiges vorgegebenes Paar k_i und k_j zufällig ein Paar (a,b) (eine Hash-Funktion) wählen, so ist das Resultatpaar (r,s) wieder ein „zufälliges“ Paar von Zahlen modulo p.

⇩

Die relative Häufigkeit, dass verschiedene Schlüssel k_i und k_j kollidieren, ist gleich der relativen Häufigkeit, dass r ≡ s mod m ist, wenn r und s zufällig modulo p gewählt werden.

Universelles Hashing

Wie viele Zahlen s kleiner als p mit $s \neq r$ und $s \equiv r \bmod m$ gibt es für eine beliebige vorgegebene Zahl $r < p$.

$$\lceil p/m \rceil - 1 \leq ((p+m-1)/m) - 1 \leq (p-1)/m$$ ⇒

Die relative Häufigkeit, dass s mit r modulo m kollidiert, ist höchstens

$$\frac{p-1}{(p-1)m} = \frac{1}{m}$$

Daher gilt für jedes beliebige Paar $k_i, k_j \in Z_p$, dass die relative Häufigkeit einer Kollision, d.h. $h(k_i) = h(k_j)$ kleiner oder gleich 1/m ist

■

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