Algorithmen und Datenstrukturen

Assoziation von Objekten, Wörterbücher, Hashing
Teil b

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Felix Kuhr (Übungen)

sowie viele Tutoren

Dictionaries: Assoziation von Objekten

- Basisoperationen (Suchen, Einfügen, Löschen)
- Güte des Hashverfahrens beeinflusst durch
  - Hashfunktion
  - Dynamisches Wachsen (und Schrumpfen)
  - Verfahren zur Kollisionsbehandlung
    - Verkettete Liste (geschlossene Adressierung)
    - Offene Adressierung
      - Lineares/Quadratisches Sondieren/Doppel-Hashing
  - Füllfaktor
- Statistisches vs. dynamisches Hashing
- Universelles Hashing

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

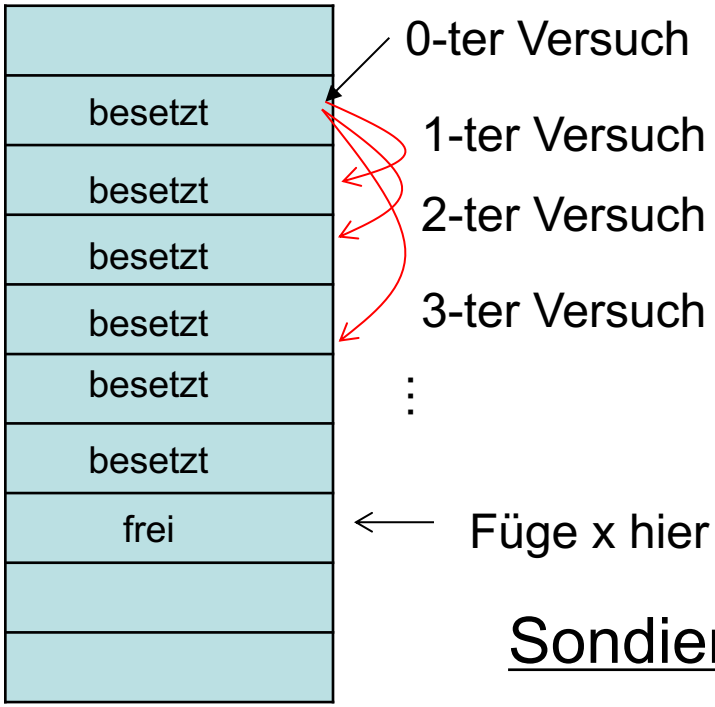
Offene Adressierung

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

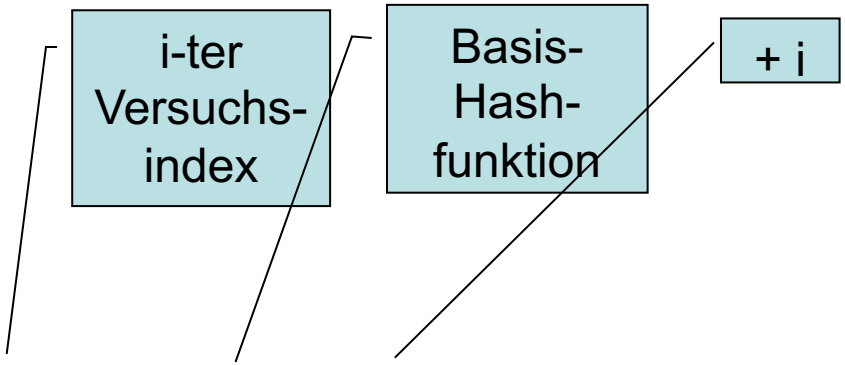
- Hashfunktion
  - $h_i(x) = (h(x) + f(i)) \bmod m$
  - $f(0) = 0$ Position des 0-ten Versuches
  - $f(i)$ „Distanz des i-ten Versuches relativ zum 0-ten Versuch“

Einfügung von x: Lineares Sondieren

Linear probing:



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



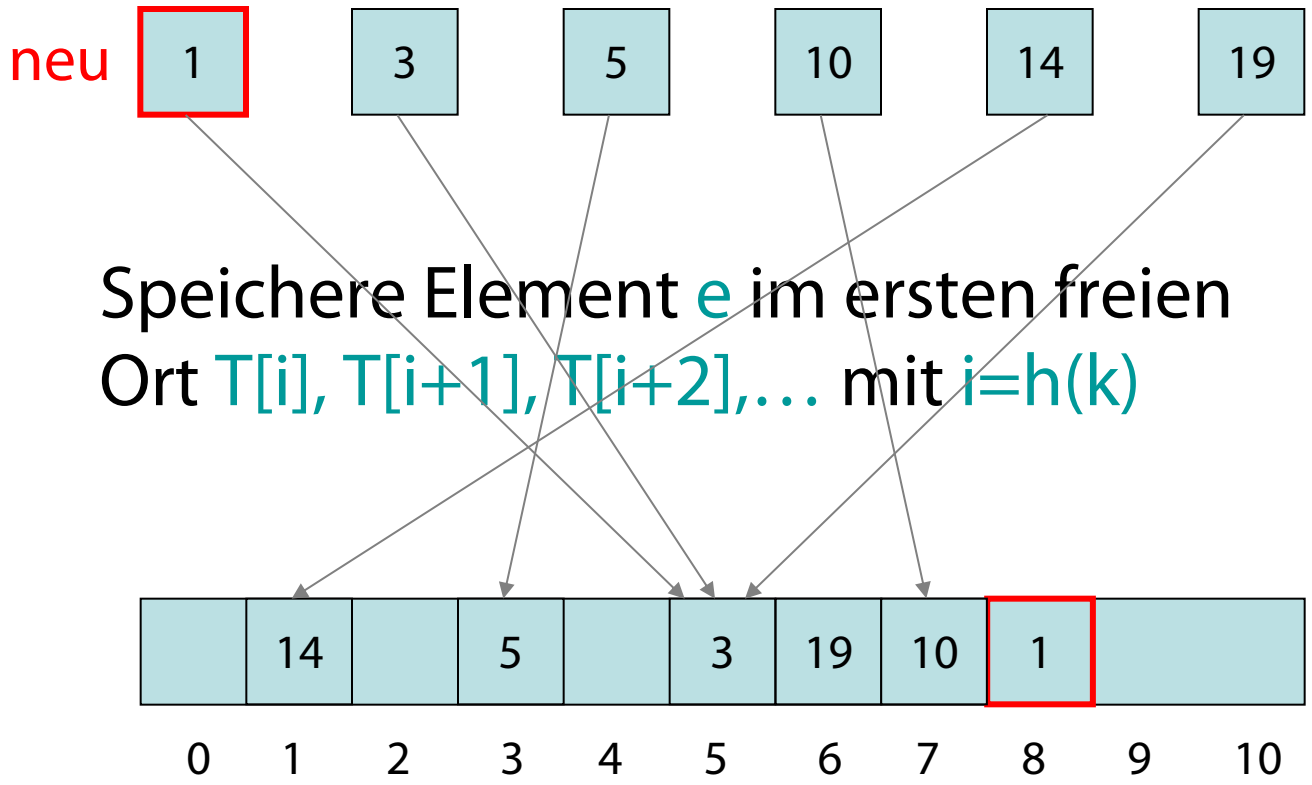
$h_i(x) = (h(x) + i) \bmod m$

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 pairs (key, Element)    // m>n

procedure insert(k, e, d:Dictionary)
    T := internalRepr(d); i := h(k)
    while T[i]<>⊥ & first(T[i])<>k do
       i := (i+1) mod length(d)
    T[i] := (k, e)

function lookup(k, d:Dictionary)
    i:=h(k); T:=internalRepr(d)
    while T[i]<>⊥ & first(T[i])<>k do
       i:=(i+1) mod length(d)
    return second(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 $i=h(k)$ und aktueller Position j gilt

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

Sei D die erwartete Suchdauer
(“gemittelte“ Anzahl von Sondierungen)

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

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

- Wir modellieren die Anzahl der Sondierungen als Zufallsvariable X mit Wertebereich $\mathbb{N}$

Beweis [AuD M. Hofmann LMU 06]

Dann ist

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

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

Mindestens i Sondierungsversuche finden statt

Kollisions-wahrscheinlichkeit

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

Analyse der erfolgreichen Suche

Die beim Aufsuchen des Schlüssels durchlaufene Sondierungsfolge ist dieselbe wie die beim Einfügen durchlaufene.

Die Länge dieser Folge für den als $i+1$-ter eingefügten Schlüssel ist im Mittel beschränkt durch $1/(1-i/m) = m/(m-i)$. (Wg. vorherigen Satzes!)

Gemittelt über alle Schlüssel, die eingefügt wurden, erhält man also

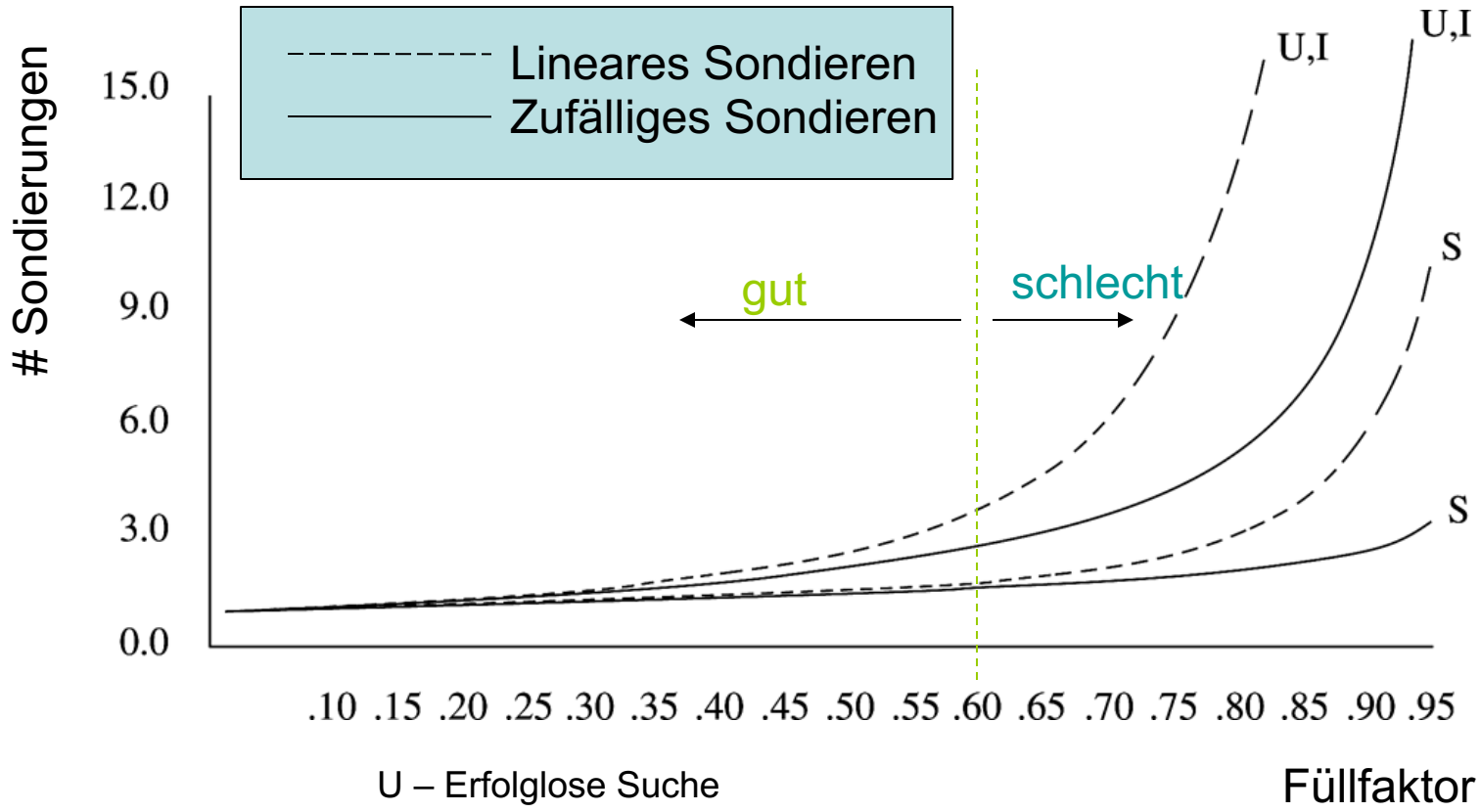
$$\begin{aligned}
\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\
&= \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \\
&\leq \frac{1}{\alpha}\int_{m-n}^{m}(1/x)\,dx \\
&= \frac{1}{\alpha}\ln\frac{m}{m-n} \\
&= \frac{1}{\alpha}\ln\frac{1}{1-\alpha}\,.
\end{aligned}$$

[AuD M. Hofmann LMU 06] und CLRS

Zufälliges Sondieren

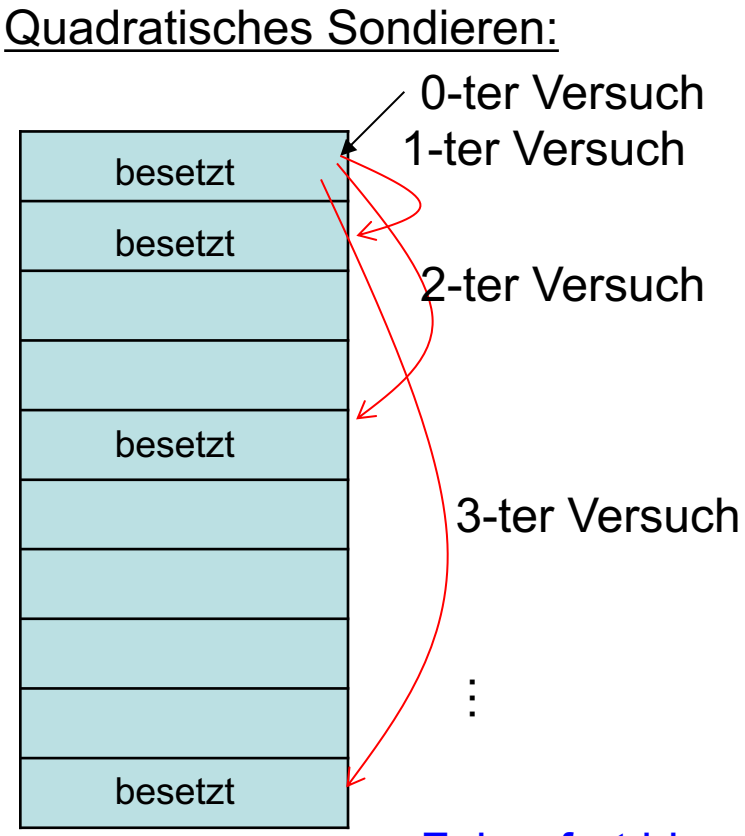
- Wähle den jeweils nächsten Feldindex schrittweise aus einer Zufallsfolge, die reproduzierbar aus dem Schlüsselwert gewonnen werden muss
  - Rechenaufwendig

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 Quadr. Sondierung

- 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

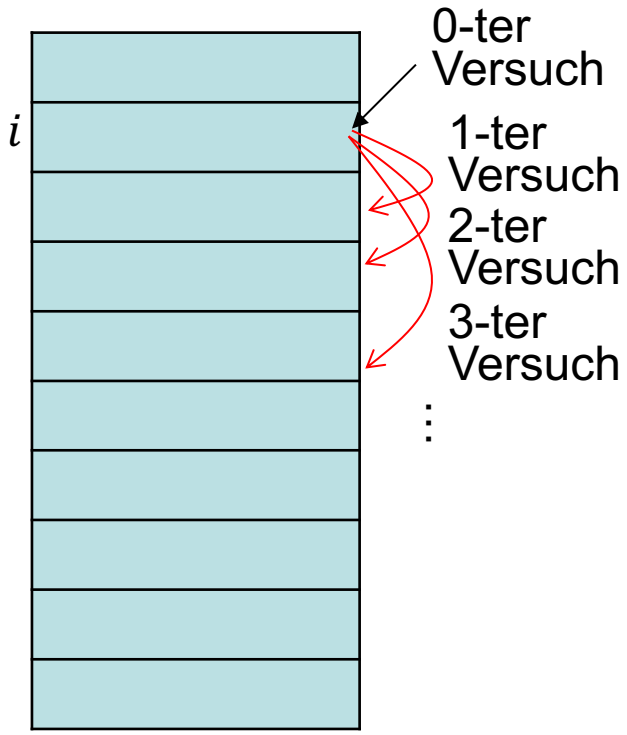
Analyse Quadratisches Sondieren

- 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

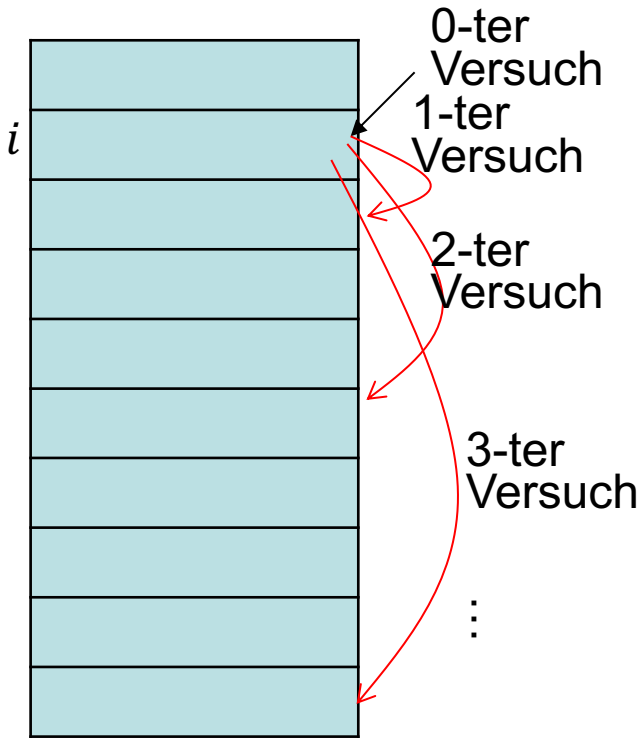
Review Hashing

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

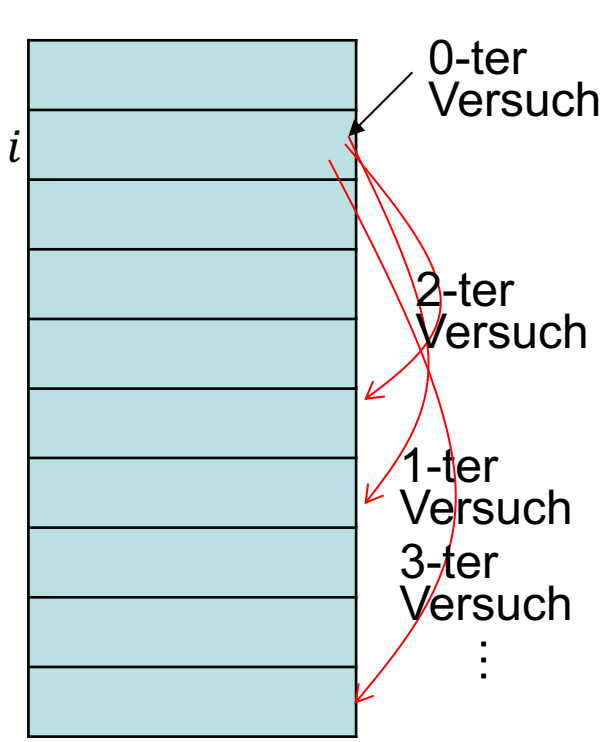
Lineares Sondieren:



Quadratisches Sondieren:



Doppel-Hashing*:



*(bestimmt mit einer zweiten Hashfunktion)

$f(i) = i$

$f(i) = i^2$

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

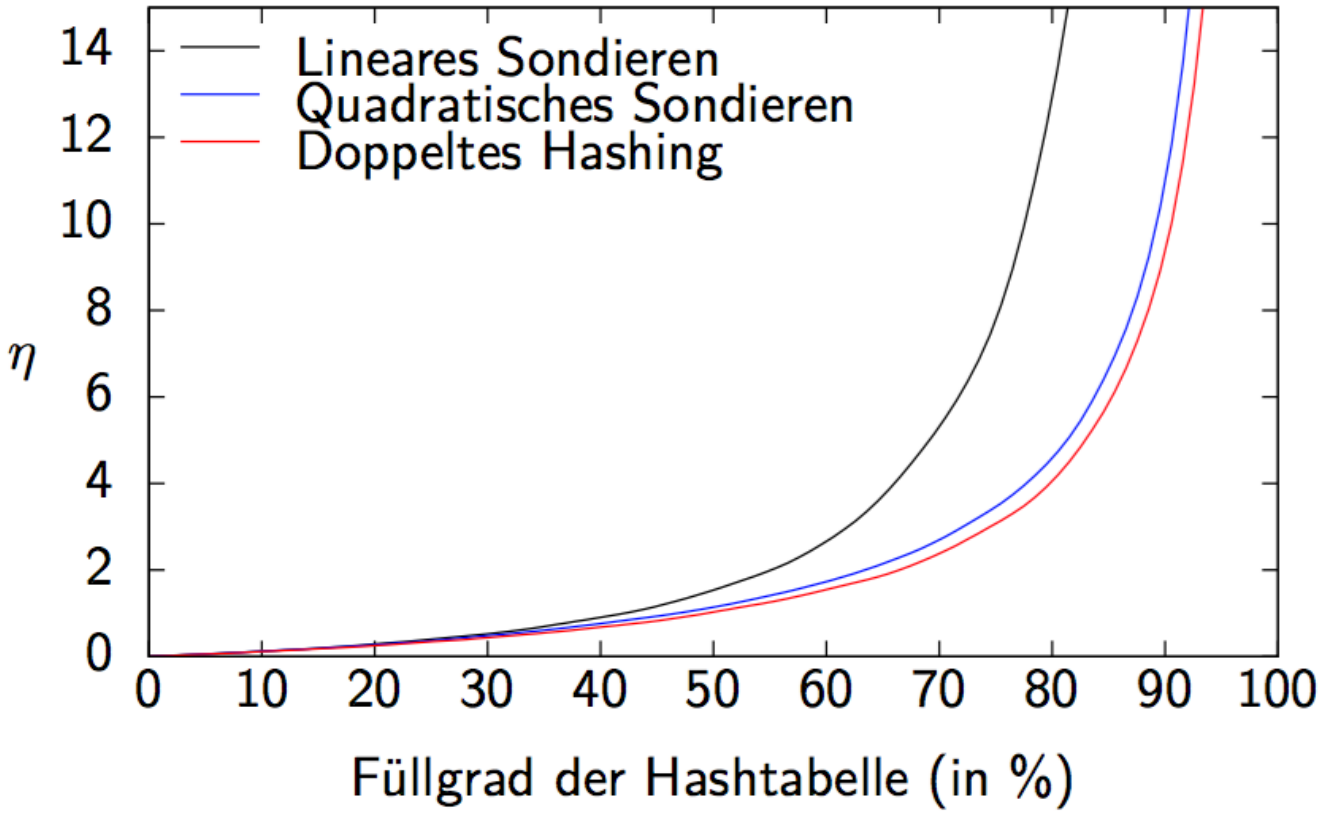
Doppel-Hashing

- Gute Wahl von h‘?
  - Sollte niemals 0 ergeben
  - h‘ (x) = x mod (m-2) + 1

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

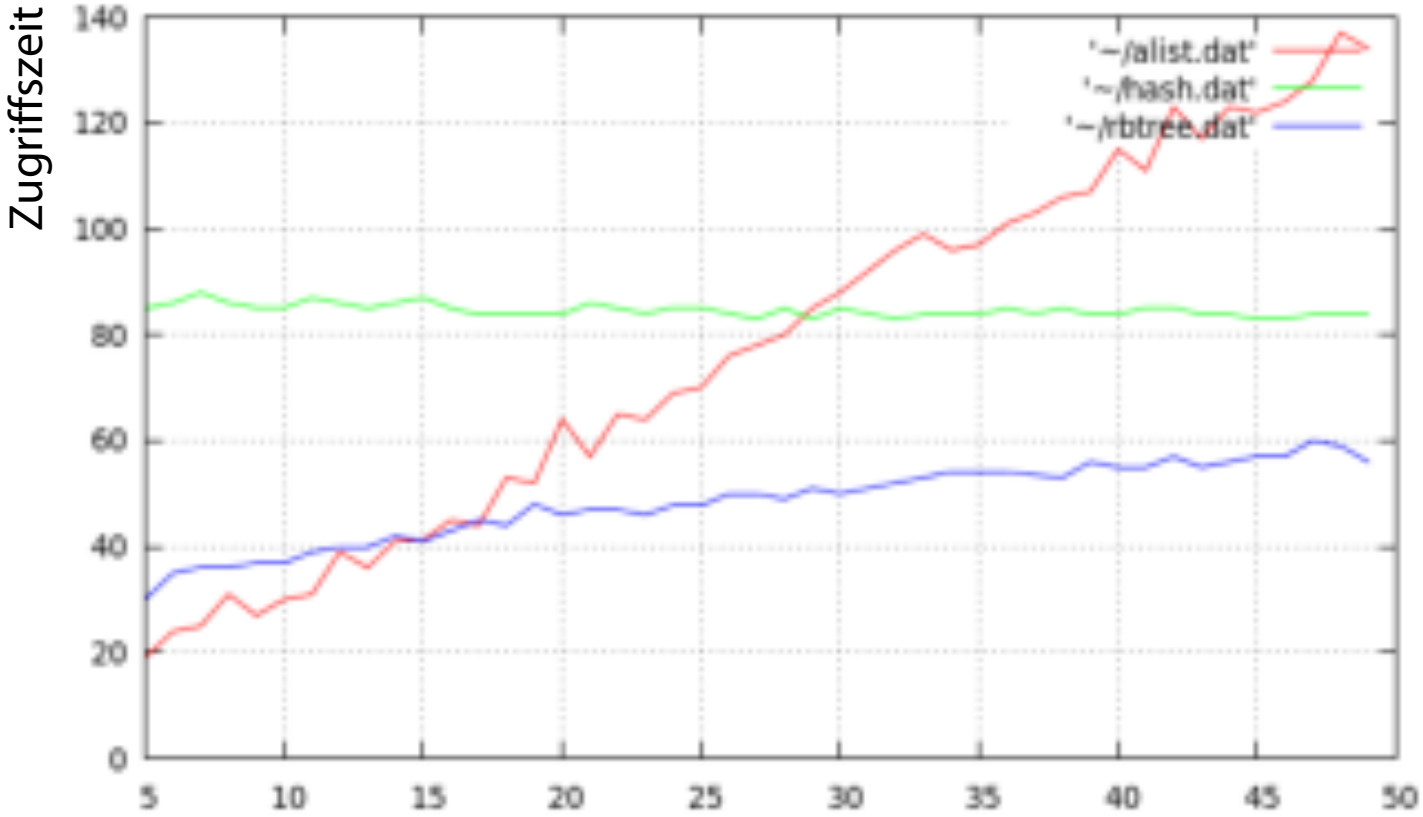
Praktische Effizienz von doppeltem Hashing

- 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)

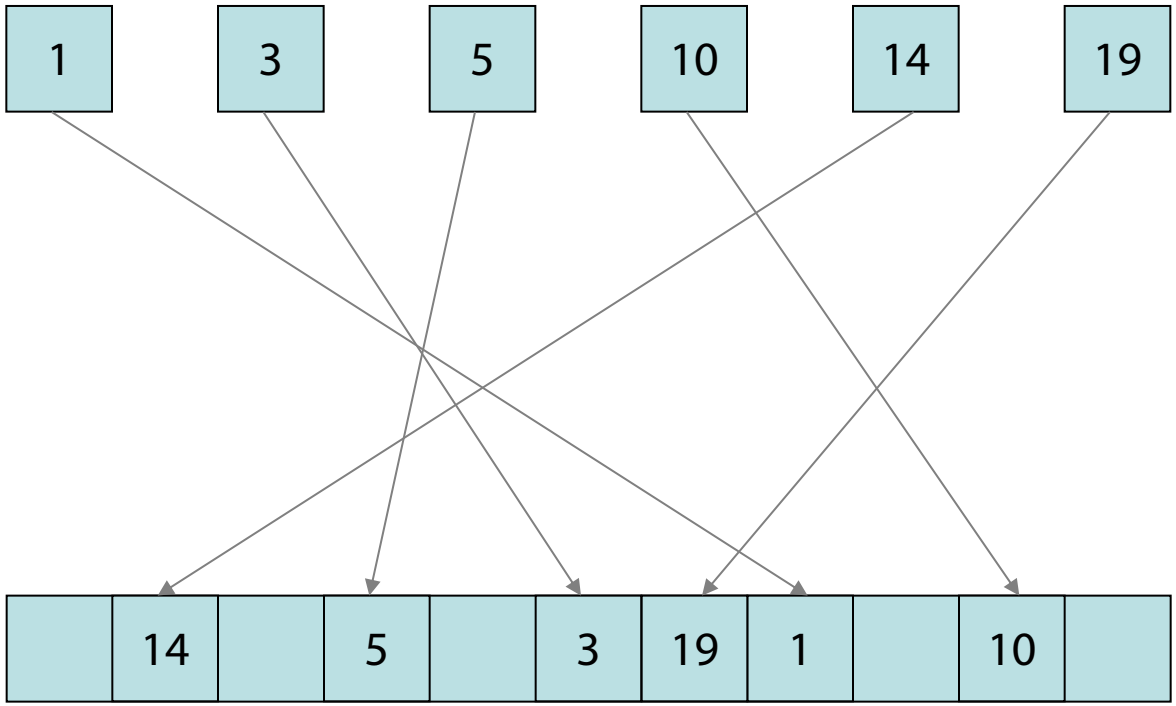


Überblick: Assoziation von Objekten

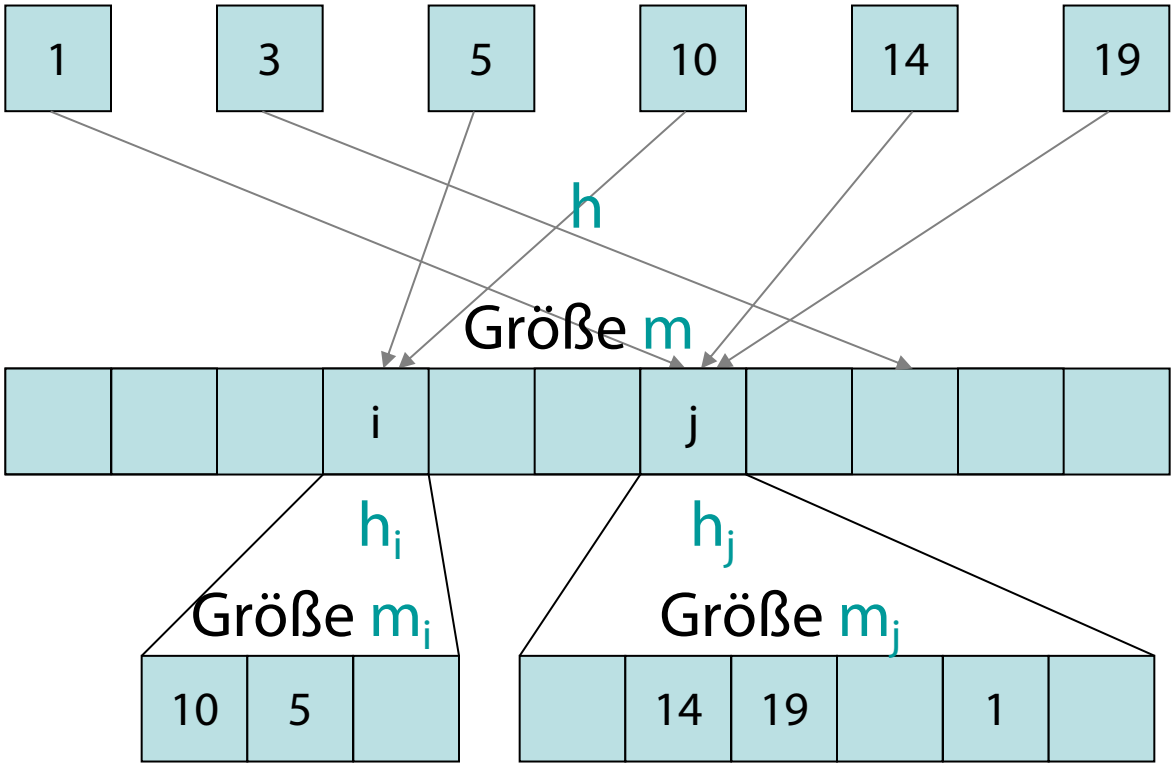
- Basisoperationen (Suchen, Einfügen, Löschen)
- Güte des Hashverfahrens beeinflusst durch
  - Hashfunktion
  - Dynamisches Wachsen (und Schrumpfen)
  - Verfahren zur Kollisionsbehandlung
    - Verkettete Liste (geschlossene Adressierung)
    - Offene Adressierung
      - Lineares/Quadratisches Sondieren/Doppel-Hashing
  - Füllfaktor
- Statistisches vs. dynamisches Hashen
- Universelles Hashing

Statisches Wörterbuch

Ziel: perfekte Hashtabelle



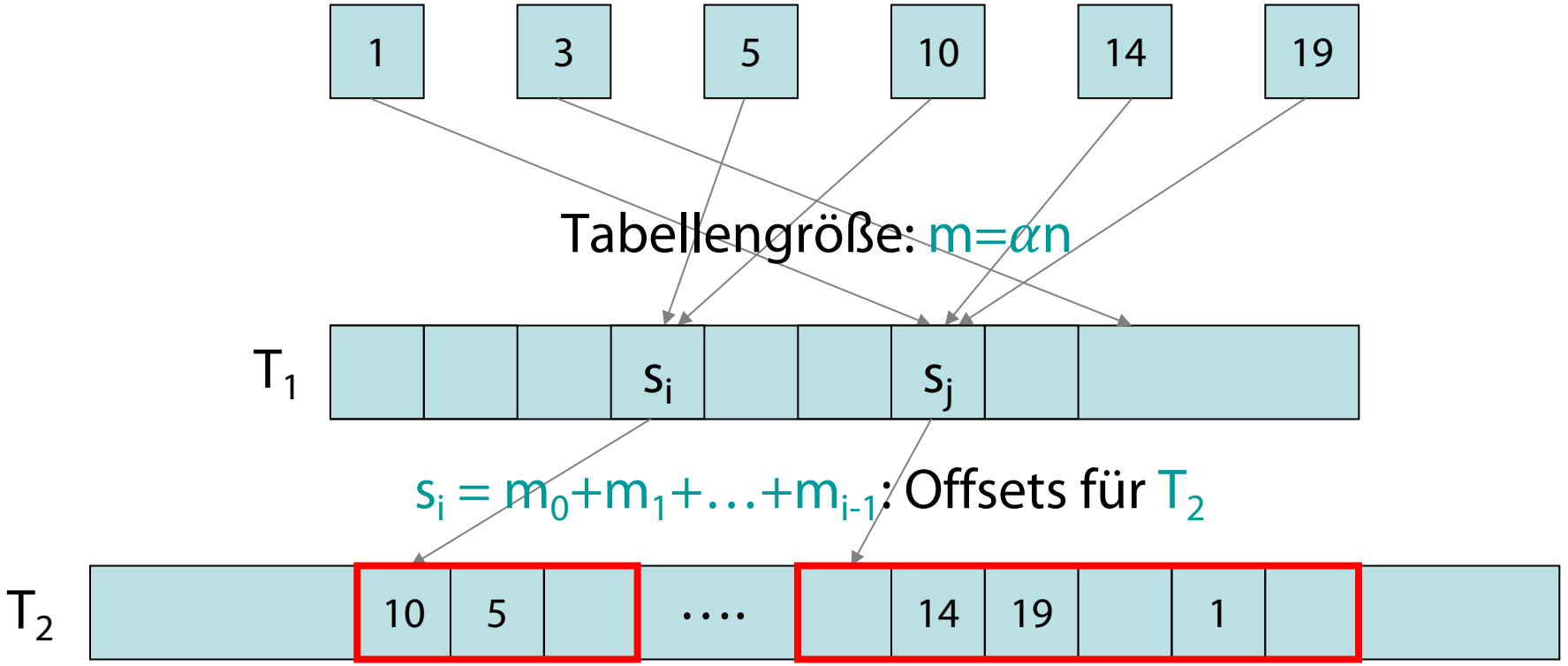
Statisches Wörterbuch (FKS-Hashing)



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 Dictionary **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: $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$
- Wir sprechen auch von einer Familie von Hashfunktionen
- Die Funktionen in H haben Parameter a, b
- Wie werden a und b bestimmt?

[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 ∈ Z_p^* und b ∈ 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**.

Zusammenfassung: Assoziation von Objekten

- Basisoperationen (Suchen, Einfügen, Löschen)
- Güte des Hashverfahrens beeinflusst durch
  - Hashfunktion
  - Dynamisches Wachsen (und Schrumpfen)
  - Verfahren zur Kollisionsbehandlung
    - Verkettete Liste (geschlossene Adressierung)
    - Offene Adressierung
      - Lineares/Quadratisches Sondieren/Doppel-Hashing
  - Füllfaktor
- Statistisches vs. dynamisches Hashen
- Universelles Hashing