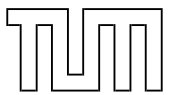
Grundlagen: Algorithmen und Datenstrukturen

Prof. Dr. Hanjo Täubig

Lehrstuhl für Effiziente Algorithmen
(Prof. Dr. Ernst W. Mayr)
Institut für Informatik
Technische Universität München

Sommersemester 2010



Übersicht

Universelles Hashing

Einfache c-universelle Hashfunktionen?

Annahme: Schlüssel sind Bitstrings einer bestimmten Länge

Annahme: Tabellengröße m ist eine Primzahl

- dann ist der Restklassenring modulo m (also $\mathbb{Z}_m$) ein Körper
- d.h. es gibt zu jedem Element außer für die Null genau ein Inverses bzgl. Multiplikation

- Sei $w = \lfloor \log_2 m \rfloor$.
- unterteile die Bitstrings der Schlüssel in Teile zu je w Bits
- Anzahl der Teile sei k
- interpretiere jeden Teil als Zahl aus dem Intervall $[0, \ldots, 2^w - 1]$
- interpretiere Schlüssel x als k-Tupel solcher Zahlen:

$$\mathbf{x} = (x_1, \ldots, x_k)$$

Familie für 1-universelles Hashing

Definiere für jeden Vektor

$$\mathbf{a} = (a_1, \ldots, a_k) \in \{0, \ldots, m-1\}^k$$

mittels Skalarprodukt

$$\mathbf{a} \cdot \mathbf{x} = \sum_{i=1}^{k} a_i x_i$$

eine Hashfunktion von der Schlüsselmenge
in die Menge der Zahlen $\{0, \ldots, m-1\}$

$$h_{\mathbf{a}}(\mathbf{x}) = \mathbf{a} \cdot \mathbf{x} \mod m$$

Familie für 1-universelles Hashing

Satz

$$H = \left\{ h_{\mathbf{a}} : \ \mathbf{a} \in \{0, \ldots, m-1\}^k \right\}$$

ist eine 1-universelle Familie von Hashfunktionen falls m prim ist.

Oder anders:
das Skalarprodukt zwischen einer Tupeldarstellung des Schlüssels und einem Zufallsvektor modulo m definiert eine gute Hashfunktion.

1-universelles Hashing

Beispiel

- Größe der Hashtabelle $m = 17$

$\Rightarrow$ Schlüssel unterteilt in k Teile mit $w = \lfloor \log_2 m \rfloor = 4$ Bits, z.B. $k = 4$

- Schlüssel sind also 4-Tupel von Integers aus dem Intervall $[0, 2^4 - 1] = \{0, \ldots, 15\}$, z.B. $\mathbf{x} = (11, 7, 4, 3)$
- Die Hashfunktion wird auch durch ein 4-Tupel von Integers, aber aus dem Intervall $[0, 17 - 1] = \{0, \ldots, 16\}$ spezifiziert
- z.B. $\mathbf{a} = (2, 4, 7, 16)$

$\Rightarrow$ Hashfunktion:

$$h_{\mathbf{a}}(\mathbf{x}) = (2x_1 + 4x_2 + 7x_3 + 16x_4) \bmod 17$$

$$h_{\mathbf{a}}(\mathbf{x}) = (2 \cdot 11 + 4 \cdot 7 + 7 \cdot 4 + 16 \cdot 3) \bmod 17 = 7$$

Eindeutiges a_j

Beweis.

- Betrachte zwei beliebige verschiedene Schlüssel $\mathbf{x} = \{x_1, \ldots, x_k\}$ und $\mathbf{y} = \{y_1, \ldots, y_k\}$
- Wie groß ist $\Pr[h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})]$?
- Sei j ein Index (von evt. mehreren möglichen) mit $x_j \neq y_j$ (muss es geben, sonst wär $\mathbf{x} = \mathbf{y}$)

$\Rightarrow$ $(x_j - y_j) \not\equiv 0 \bmod m$

$\Rightarrow$ gegeben Primzahl m und Zahlen $x_j, y_j, b \in \{0, \ldots m-1\}$ hat jede Gleichung der Form

$$a_j(x_j - y_j) \equiv b \bmod m$$

eine eindeutige Lösung: $a_j \equiv (x_j - y_j)^{-1} b \bmod m$, wobei $(x_j - y_j)^{-1}$ das multiplikative Inverse von $(x_j - y_j)$ ist

Wann wird $h(\mathbf{x}) = h(\mathbf{y})$?

Beweis.

Wenn man alle Variablen a_i außer a_j festlegt, gibt es exakt eine Wahl für a_j, so dass $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$, denn

$$\begin{aligned}
h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y}) \quad &\Leftrightarrow \quad \sum_{i=1}^{k} a_i x_i \equiv \sum_{i=1}^{k} a_i y_i && \mod m \\
&\Leftrightarrow \quad a_j(x_j - y_j) \equiv \sum_{i \neq j} a_i (y_i - x_i) && \mod m \\
&\Leftrightarrow \quad a_j \equiv (x_j - y_j)^{-1} \sum_{i \neq j} a_i (y_i - x_i) && \mod m
\end{aligned}$$

Wie oft wird $h(\mathbf{x}) = h(\mathbf{y})$?

Beweis.

- Es gibt m^{k-1} Möglichkeiten, Werte für die Variablen a_i mit $i \neq j$ zu wählen.
- Für jede solche Wahl gibt es genau eine Wahl für a_j, so dass $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$.
- Für $\mathbf{a}$ gibt es insgesamt m^k Auswahlmöglichkeiten.
- Also

$$\Pr[h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})] = \frac{m^{k-1}}{m^k} = \frac{1}{m}$$

□

Universelles Hashing

Definiere für $a \in \{0, \ldots, m-1\}$ die Hashfunktion

$$h'_a(\mathbf{x}) = \sum_{i=1}^{k} a^{i-1} x_i \bmod m$$

(mit $x_i \in \{0, \ldots, m-1\}$)

Satz

Für jede Primzahl m ist

$$H' = \{h'_a : \ a \in \{0, \ldots, m-1\}\}$$

eine $(k-1)$-universelle Familie von Hashfunktionen.

Universelles Hashing

Beweisidee:

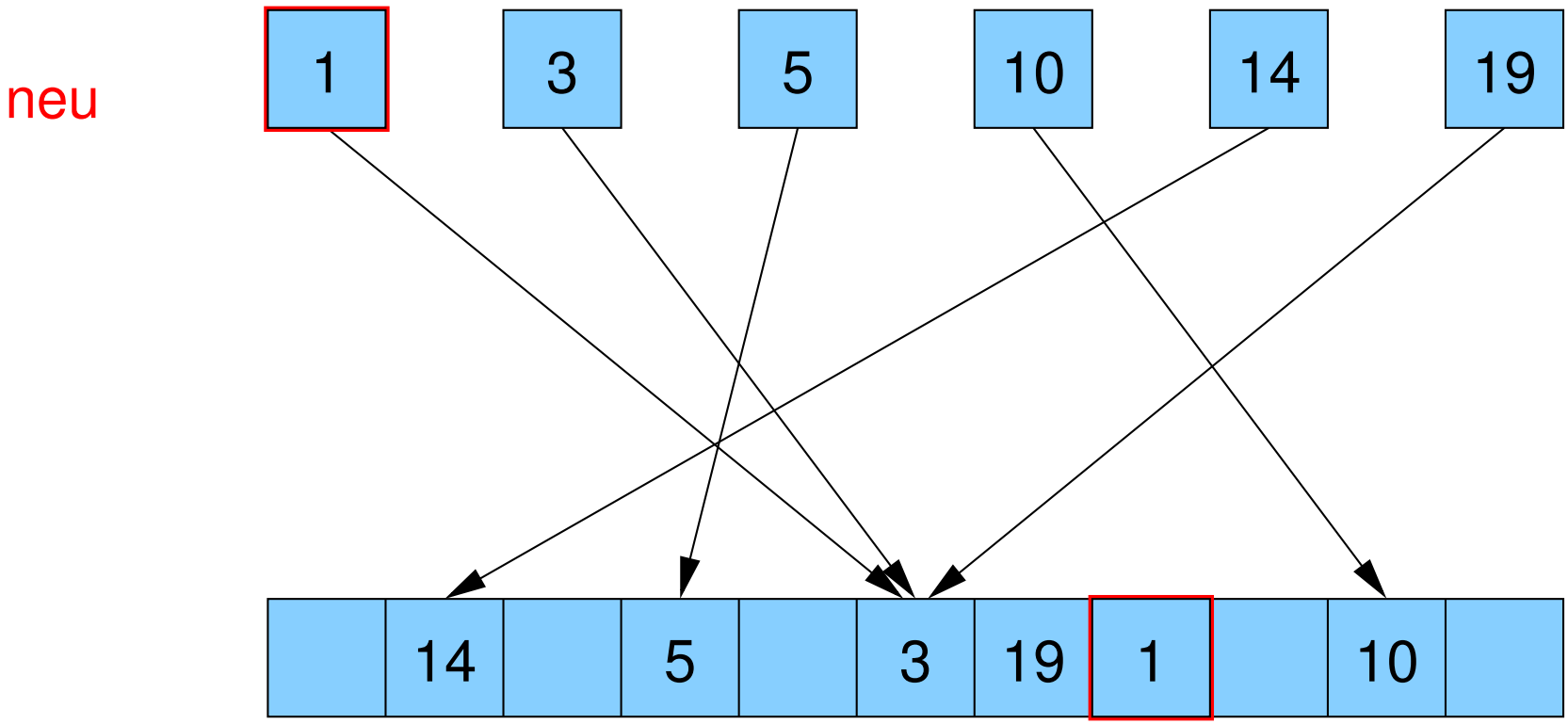
Für Schlüssel $\mathbf{x} \neq \mathbf{y}$ ergibt sich folgende Gleichung:

$$\begin{aligned} h'_a(\mathbf{x}) &\equiv h'_a(\mathbf{y}) \bmod m \\ h'_a(\mathbf{x}) - h'_a(\mathbf{y}) &\equiv 0 \bmod m \\ \sum_{i=1}^{k} a^{i-1}(x_i - y_i) &\equiv 0 \bmod m \end{aligned}$$

Diese Gleichung ist in der Anzahl der Lösungen in a durch den Grad des Polynoms beschränkt (Fundamentalsatz der Algebra), also durch $k - 1$.

Dynamisches Wörterbuch

Hashing with Linear Probing:



Speichere Element e im ersten freien Ort T[i], T[$i+1$], T[$i+2$], ... mit $i == h(\text{key}(e))$
(Ziel: Folgen besetzter Positionen möglichst kurz)

Hashing with Linear Probing

```
Elem [ ] T;                    // Feld sollte genügend groß sein

void insert(Elem e) {
   i = h(key(e));
   while (T[i] != null && T[i] != e)
      i = (i+1) % m;
   T[i] = e;
}

void find(Key k) {
   i = h(k);
   while (T[i] != null && key(T[i]) != k)
      i = (i+1) % m;
   return T[i];
}
```

Hashing with Linear Probing

Vorteil:

Es werden im Gegensatz zu Hashing with Chaining (oder auch im Gegensatz zu anderen Probing-Varianten) nur zusammenhängende Speicherzellen betrachtet.

$\Rightarrow$ Cache-Effizienz!

Hashing with Linear Probing

Problem: Löschen von Elementen

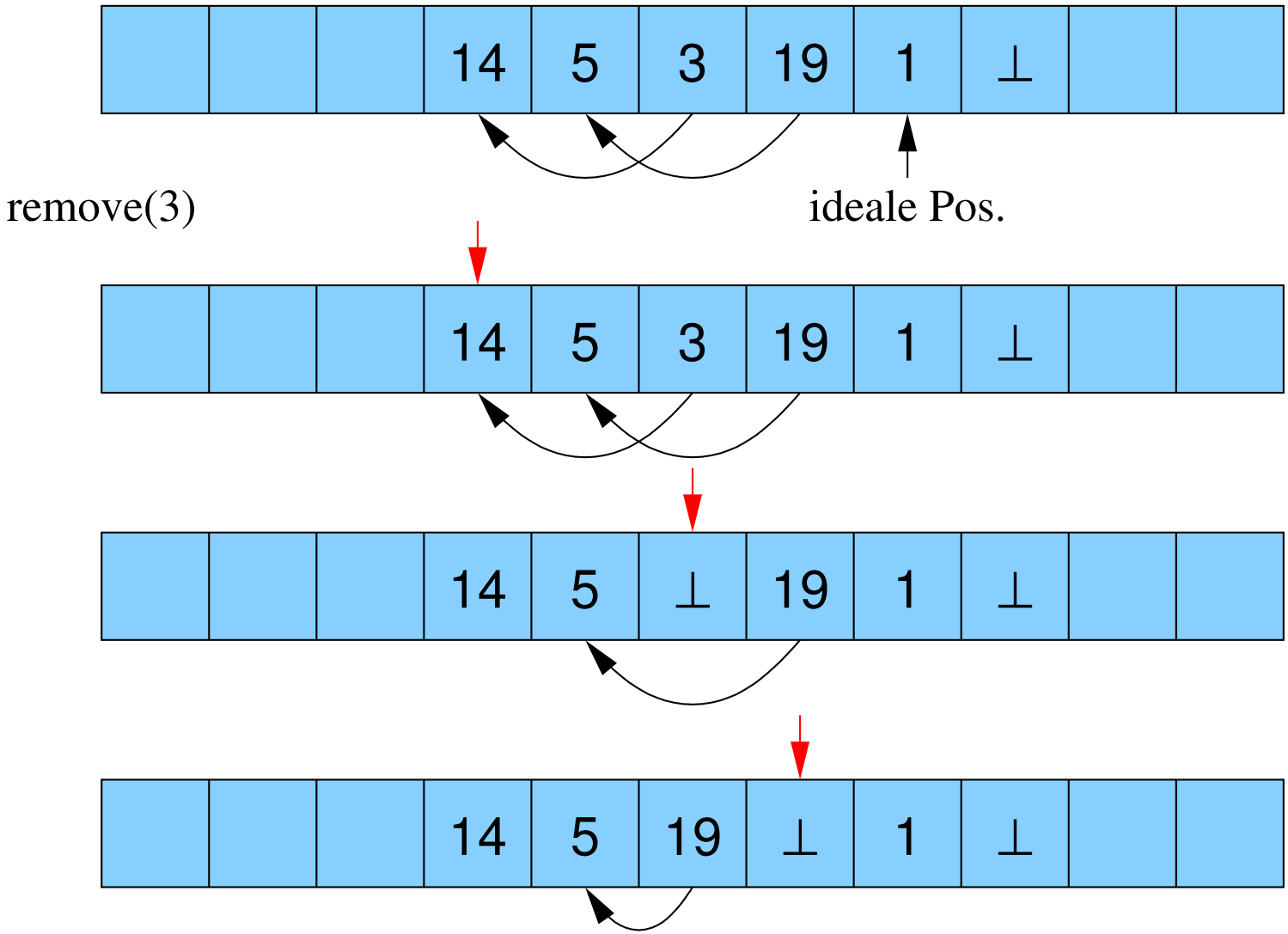
1. Löschen verbieten
2. Markiere Positionen als gelöscht
   (mit speziellem Zeichen $\neq \perp$)
   Suche endet bei $\perp$, aber nicht bei markierten Zellen

   Problem: Anzahl echt freier Zellen sinkt monoton
   $\Rightarrow$ Suche wird evt. langsam oder periodische Reorganisation
3. Invariante sicherstellen:

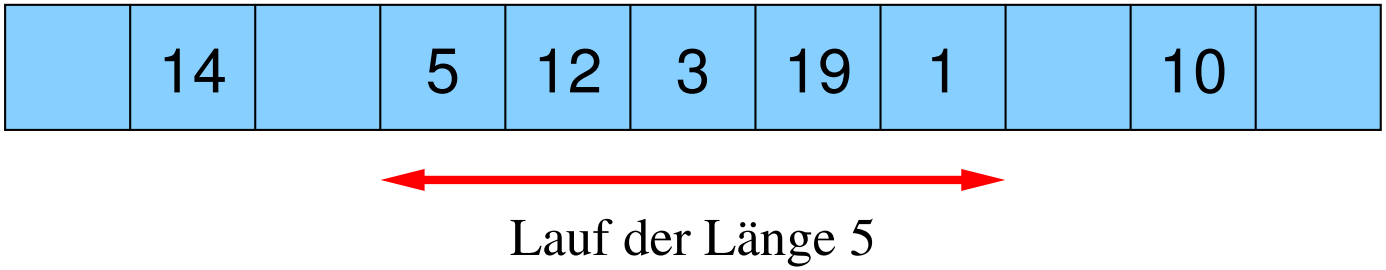
   Für jedes $e \in S$ mit idealer Position $i = h(key(e))$ und aktueller Position j gilt:
   T[i], T[i+1],. . . ,T[j] sind besetzt

Hashing with Linear Probing

Löschen / Aufrechterhaltung der Invariante



Hashing with Linear Probing



Satz

Wenn n Elemente in einer Hashtabelle T der Größe $m > 2en$ mittels einer zufälligen Hashfunktion h gespeichert werden, dann ist für jedes $T[i]$ die erwartete Länge eines Laufes in T, der $T[i]$ enthält, $O(1)$.

(e ist hier die Eulersche Zahl)

Hashing with Linear Probing

Beweis.

- n: Anzahl der Elemente
- $m > 2en$: Größe der Hashtabelle
- Lauf der Länge k: k aufeinanderfolgende besetzte Zellen
- Anzahl Möglichkeiten zur Wahl von k Elementen: $\binom{n}{k} \leq \left(\frac{en}{k}\right)^k$
- Wahrscheinlichkeit, dass k Hashwerte genau einen Lauf der Länge k an einer bestimmten Stelle ergeben: $\frac{k^{\underline{k}}}{m^k} < \left(\frac{k}{m}\right)^k$
- Es gibt aber k verschiedene Anfangspositionen des Laufs der Länge k, so dass $T[i]$ darin liegt.

$\Rightarrow$ $p_k = \Pr[T[i] \text{ in Lauf der Länge } k] < \left(\frac{en}{k}\right)^k k \left(\frac{k}{m}\right)^k < k\left(\frac{1}{2}\right)^k$

Hashing with Linear Probing

Beweis.

- Erwartete Länge des Laufs:

$$\begin{aligned}\mathbb{E}[\text{Länge des Laufs um } T[i]] &= \sum_{k\geq 1} k \cdot p_k \\ &< \sum_{k\geq 1} k^2 \left(\frac{1}{2}\right)^k \\ &= O(1)\end{aligned}$$

D.h. also die erwartete Laufzeit der Operationen insert, remove und find ist eine Konstante.

□