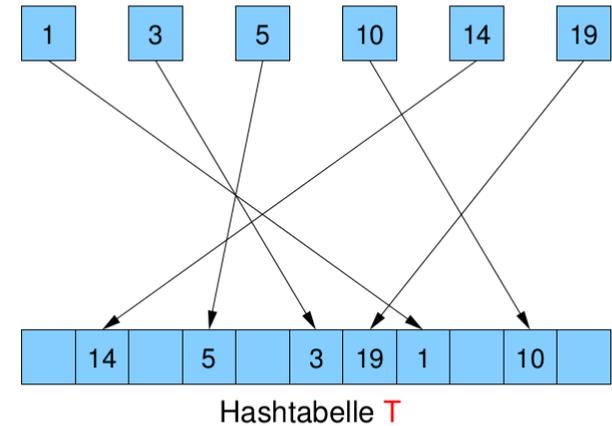


Hashfunktion und Hashtabelle



Hashfunktion h :
 Key $\mapsto \{0, \dots, m - 1\}$
 $|\text{Key}| = N$
 gespeicherte
 Elemente: n

Script generated by TTT

Title: TÄubig: GAD (14.05.2013)
 Date: Tue May 14 14:16:04 CEST 2013
 Duration: 89:23 min
 Pages: 39

Hashfunktion

Anforderungen:

- schneller Zugriff (Zeiteffizienz)
 - platzsparend (Speichereffizienz)
 (z.B. surjektive Abbildung möglicher Schlüssel auf die Adressen)
 - **gute Streuung** bzw. Verteilung der Elemente über die ganze Tabelle
 - Idealfall: Element e direkt in Tabelleneintrag $t[h(\text{key}(e))]$
- ⇒ find, insert und remove in **konstanter Zeit**
 (genauer: plus Zeit für Berechnung der Hashfunktion)

Hashing

Annahme: perfekte Streuung

```
insert(Elem e) {
    T[h(key(e))] = e;
}
```

```
remove(Key k) {
    T[h(k)] = null;
}
```

```
Elem find(Key k) {
    return T[h(k)];
}
```

statisches Wörterbuch: nur find
 dynamisches Wörterbuch: insert, remove und find

Kollisionen

In der Praxis:

- perfekte Zuordnung zwischen den gespeicherten Schlüsseln und den Adressen der Tabelle nur bei statischem Array möglich
- leere Tabelleneinträge
- Schlüssel mit gleicher Adresse (Kollisionen)

Wie wahrscheinlich ist eine Kollision?

- Geburtstagsparadoxon: In einer Menge von 23 zufällig ausgewählten Personen gibt es mit Wahrscheinlichkeit > 50% zwei Leute, die am gleichen Tag Geburtstag feiern.
- Bei zufälliger Abbildung von 23 Schlüsseln auf die Adressen einer Hashtabelle der Größe 365 gibt es mit Wahrscheinlichkeit > 50% eine Kollision.

Wahrscheinlichkeit von Kollisionen



- Hilfsmittel: $\forall x \in \mathbb{R} : 1 + x \leq \sum_{i=0}^{\infty} \frac{x^i}{i!} = e^x$
- $\Rightarrow \forall x \in \mathbb{R} : \ln(1 + x) \leq x$ (da $\ln(x)$ monoton wachsend ist)
- $\text{Pr}[\text{keine Kollision beim } i\text{-ten Schlüssel}] = \frac{m - (i - 1)}{m}$ für $i \in [1 \dots n]$

$$\begin{aligned} \text{Pr}[\text{keine Kollision}] &= \prod_{i=1}^n \frac{m - (i - 1)}{m} = \prod_{i=0}^{n-1} \left(1 - \frac{i}{m}\right) \\ &= e^{\left[\sum_{i=0}^{n-1} \ln\left(1 - \frac{i}{m}\right)\right]} \leq e^{\left[\sum_{i=0}^{n-1} \left(-\frac{i}{m}\right)\right]} = e^{-\frac{n(n-1)}{2m}} \end{aligned}$$

(da e^x monoton wachsend ist)

\Rightarrow Bei gleichverteilt zufälliger Hashposition für jeden Schlüssel tritt für $n \in \omega(\sqrt{m})$ mit Wahrscheinlichkeit $1 - o(1)$ mindestens eine Kollision auf.

m ∈ o(n²)

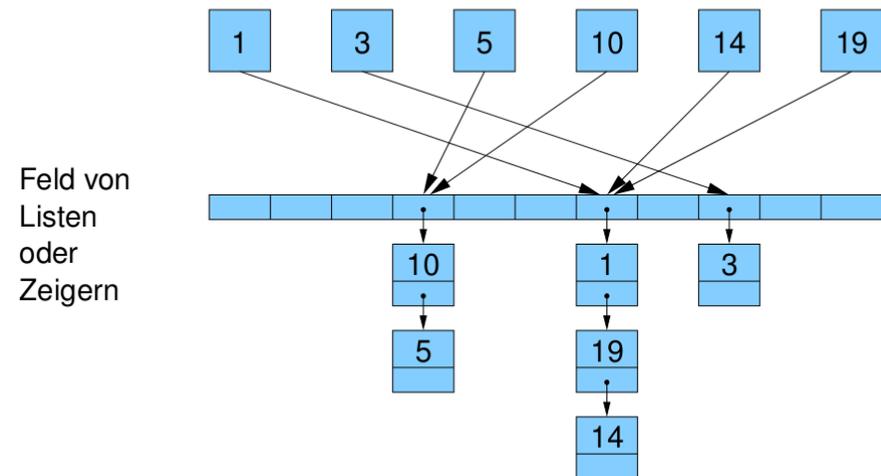
Übersicht

5 Hashing

- Hashtabellen
- Hashing with Chaining
- Universelles Hashing
- Hashing with Linear Probing
- Anpassung der Tabellengröße
- Perfektes Hashing
- Diskussion / Alternativen

Dynamisches Wörterbuch

Hashing with Chaining:



unsortierte verkettete Listen
(Ziel: Listen möglichst kurz)

Dynamisches Wörterbuch

Hashing with **Chaining**:

List<Elem>[m] T:

```
insert(Elem e) {
  T[h(key(e))].insert(e);
}
```

```
remove(Key k) {
  T[h(k)].remove(k);
}
```

```
find(Key k) {
  T[h(k)].find(k);
}
```

Hashing with Chaining

- Platzverbrauch: $O(n + m)$
- insert benötigt konstante Zeit
- remove und find müssen u.U. eine ganze Liste scannen
- im worst case sind alle Elemente in dieser Liste

⇒ im worst case ist Hashing with chaining nicht besser als eine normale Liste

Hashing with Chaining

Gibt es Hashfunktionen, die garantieren, dass alle Listen kurz sind?

- nein, für jede Hashfunktion gibt es eine Adresse, der mindestens $\frac{N}{m}$ mögliche Schlüssel zugeordnet sind (erweitertes **Schubfachprinzip** / pigeonhole principle)
- Meistens ist $n < \frac{N}{m}$ (weil N riesig ist).
- In diesem Fall kann die Suche zum Scan aller Elemente entarten.

⇒ Auswege

- Average-case-Analyse
- Randomisierung
- Änderung des Algorithmus (z.B. Hashfunktion abhängig von aktuellen Schlüsseln)

Hashing with Chaining

Betrachte als Hashfunktionsmenge die Menge aller Funktionen, die die Schlüsselmenge (mit Kardinalität N) auf die Zahlen $0, \dots, m - 1$ abbilden.

Satz

Falls n Elemente in einer Hashtabelle der Größe m mittels einer zufälligen Hashfunktion gespeichert werden, dann ist die erwartete Laufzeit von remove bzw. find in $O(1 + n/m)$.

Unrealistisch: es gibt m^N solche Funktionen und man braucht $\sqrt{\log_2(m^N)} = N \log_2 m$ Bits, um eine Funktion zu spezifizieren.

⇒ widerspricht dem Ziel, den Speicherverbrauch von N auf n zu senken!

Hashing with Chaining

Beweis.

- Betrachte feste Position $i = h(k)$ bei $\text{remove}(k)$ oder $\text{find}(k)$
- Laufzeit ist Konstante plus Zeit für Scan der Liste also $O(1 + \mathbb{E}[X])$, wobei X Zufallsvariable für Länge von $T[i]$

Hashing with Chaining

Beweis.

- Betrachte feste Position $i = h(k)$ bei $\text{remove}(k)$ oder $\text{find}(k)$
- Laufzeit ist Konstante plus Zeit für Scan der Liste also $O(1 + \mathbb{E}[X])$, wobei X Zufallsvariable für Länge von $T[i]$
- Zufallsvariable $X_e \in \{0, 1\}$ für jedes $e \in S$
- $X_e = 1 \Leftrightarrow h(\text{key}(e)) = i$
- Listenlänge $X = \sum_{e \in S} X_e$

Hashing with Chaining

Beweis.

- Betrachte feste Position $i = h(k)$ bei $\text{remove}(k)$ oder $\text{find}(k)$
- Laufzeit ist Konstante plus Zeit für Scan der Liste also $O(1 + \mathbb{E}[X])$, wobei X Zufallsvariable für Länge von $T[i]$
- Zufallsvariable $X_e \in \{0, 1\}$ für jedes $e \in S$
- $X_e = 1 \Leftrightarrow h(\text{key}(e)) = i$
- Listenlänge $X = \sum_{e \in S} X_e$
- Erwartete Listenlänge $\mathbb{E}[X]$

$$\begin{aligned}
 &= \mathbb{E} \left[\sum_{e \in S} X_e \right] = \sum_{e \in S} \mathbb{E}[X_e] = \sum_{e \in S} 0 \cdot \Pr[X_e = 0] + 1 \cdot \Pr[X_e = 1] \\
 &= \sum_{e \in S} \Pr[X_e = 1] = \sum_{e \in S} 1/m = n/m
 \end{aligned}$$

□

Übersicht

- 5 Hashing
 - Hashtabellen
 - Hashing with Chaining
 - **Universelles Hashing**
 - Hashing with Linear Probing
 - Anpassung der Tabellengröße
 - Perfektes Hashing
 - Diskussion / Alternativen

c-universelle Familien von Hashfunktionen

Wie konstruiert man zufällige Hashfunktionen?

Definition

Sei c eine positive Konstante.

Eine Familie H von Hashfunktionen auf $\{0, \dots, m-1\}$ heißt **c-universell**, falls für jedes Paar $x \neq y$ von Schlüsseln gilt, dass

$$|\{h \in H : h(x) = h(y)\}| \leq \frac{c}{m} |H|.$$


c-universelle Familien von Hashfunktionen

Wie konstruiert man zufällige Hashfunktionen?

Definition

Sei c eine positive Konstante.

Eine Familie H von Hashfunktionen auf $\{0, \dots, m-1\}$ heißt **c-universell**, falls für jedes Paar $x \neq y$ von Schlüsseln gilt, dass

$$|\{h \in H : h(x) = h(y)\}| \leq \frac{c}{m} |H|.$$

D.h. bei zufälliger Auswahl der Hashfunktion $h \in H$ gilt

$$\forall \{x, y\}_{(x \neq y)} : \Pr[h(x) = h(y)] \leq \frac{c}{m}$$

1-universelle Familien nennt man **universell**.

c-Universal Hashing with Chaining

Satz

Falls n Elemente in einer Hashtabelle der Größe m mittels einer zufälligen Hashfunktion h aus einer **c-universellen** Familie gespeichert werden, dann ist die erwartete Laufzeit von `remove` bzw. `find` in $O(1 + \frac{c \cdot n}{m})$.

c-Universal Hashing with Chaining

Satz

Falls n Elemente in einer Hashtabelle der Größe m mittels einer zufälligen Hashfunktion h aus einer **c-universellen** Familie gespeichert werden, dann ist die erwartete Laufzeit von `remove` bzw. `find` in $O(1 + \frac{c \cdot n}{m})$.

Beweis.

- Betrachte festen Schlüssel k
- Zugriffszeit X ist $O(1 + \text{Länge der Liste } T[h(k)])$
- Zufallsvariable $X_e \in \{0, 1\}$ für jedes $e \in S$ zeigt an, ob e auf die gleiche Position wie k gehasht wird

c-Universal Hashing with Chaining

Beweis.

- $X_e = 1 \Leftrightarrow h(\text{key}(e)) = h(k)$
- Listenlänge $X = \sum_{e \in S} X_e$
- Erwartete Listenlänge

$$\begin{aligned} \mathbb{E}[X] &= \mathbb{E}\left[\sum_{e \in S} X_e\right] \\ &= \sum_{e \in S} \mathbb{E}[X_e] = \sum_{e \in S} 0 \cdot \Pr[X_e = 0] + 1 \cdot \Pr[X_e = 1] \\ &= \sum_{e \in S} \Pr[X_e = 1] \leq \sum_{e \in S} c/m = n \cdot c/m \end{aligned}$$

□

Familie für universelles Hashing

Definiere für jeden Vektor

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

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 $\{0, \dots, m-1\}$

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

Beispiele für c-universelles Hashing

Einfache c-universelle Hashfunktionen?

Annahme: Schlüssel sind Bitstrings einer bestimmten Länge

Wähle als Tabellengröße m eine Primzahl

\Rightarrow 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 $[0, \dots, 2^w - 1]$
- interpretiere Schlüssel x als k -Tupel solcher Zahlen:

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

Familie für universelles Hashing

Satz

Wenn m eine Primzahl ist, dann ist

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

eine [1]-universelle Familie von Hashfunktionen.

Oder anders:

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

Familie für universelles Hashing

Beispiel

- 32-Bit-Schlüssel, Hashtabellengröße $m = 269$

Familie für universelles Hashing

Beispiel

- 32-Bit-Schlüssel, Hashtabellengröße $m = 269$
- ⇒ Schlüssel unterteilt in $k = 4$ Teile mit $w = \lfloor \log_2 m \rfloor = 8$ Bits

Familie für universelles Hashing

Beispiel

- 32-Bit-Schlüssel, Hashtabellengröße $m = 269$
- ⇒ Schlüssel unterteilt in $k = 4$ Teile mit $w = \lfloor \log_2 m \rfloor = 8$ Bits
- Schlüssel sind also 4-Tupel von Integers aus dem Intervall $[0, 2^8 - 1] = \{0, \dots, 255\}$, z.B. $\mathbf{x} = (11, 7, 4, 3)$

Familie für universelles Hashing

Beispiel

- 32-Bit-Schlüssel, Hashtabellengröße $m = 269$
- ⇒ Schlüssel unterteilt in $k = 4$ Teile mit $w = \lfloor \log_2 m \rfloor = 8$ Bits
- Schlüssel sind also 4-Tupel von Integers aus dem Intervall $[0, 2^8 - 1] = \{0, \dots, 255\}$, z.B. $\mathbf{x} = (11, 7, 4, 3)$
- Die Hashfunktion wird auch durch ein 4-Tupel von Integers, aber aus dem Intervall $[0, 269 - 1] = \{0, \dots, 268\}$, spezifiziert z.B. $\mathbf{a} = (2, 4, 261, 16)$

Familie für universelles Hashing

Beispiel

- 32-Bit-Schlüssel, Hashtabellengröße $m = 269$
- ⇒ Schlüssel unterteilt in $k = 4$ Teile mit $w = \lfloor \log_2 m \rfloor = 8$ Bits
- Schlüssel sind also 4-Tupel von Integers aus dem Intervall $[0, 2^8 - 1] = \{0, \dots, 255\}$, z.B. $\mathbf{x} = (11, 7, 4, 3)$
- Die Hashfunktion wird auch durch ein 4-Tupel von Integers, aber aus dem Intervall $[0, 269 - 1] = \{0, \dots, 268\}$, spezifiziert z.B. $\mathbf{a} = (2, 4, 261, 16)$
- ⇒ Hashfunktion:

$$h_{\mathbf{a}}(\mathbf{x}) = (2x_1 + 4x_2 + 261x_3 + 16x_4) \bmod 269$$

$$h_{\mathbf{a}}(\mathbf{x}) = (2 \cdot 11 + 4 \cdot 7 + 261 \cdot 4 + 16 \cdot 3) \bmod 269 = 66$$

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

$$h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y}) \Leftrightarrow \sum_{i=1}^k a_i x_i \equiv \sum_{i=1}^k a_i y_i \pmod{m}$$

$$\Leftrightarrow a_j(x_j - y_j) \equiv \sum_{i \neq j} a_i(y_i - x_i) \pmod{m}$$

$$\Leftrightarrow a_j \equiv (x_j - y_j)^{-1} \sum_{i \neq j} a_i(y_i - x_i) \pmod{m}$$

Eindeutiges a_j

Beweis

- Betrachte zwei beliebige verschiedene Schlüssel $\mathbf{x} = \{x_1, \dots, x_k\}$ und $\mathbf{y} = \{y_1, \dots, y_k\}$
- 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 wäre $\mathbf{x} = \mathbf{y}$)
- ⇒ $(x_j - y_j) \not\equiv 0 \pmod{m}$
d.h., es gibt genau ein multiplikatives Inverses $(x_j - y_j)^{-1}$
- ⇒ gegeben Primzahl m und Zahlen $x_j, y_j, b \in \{0, \dots, m-1\}$ hat jede Gleichung der Form

$$a_j(x_j - y_j) \equiv b \pmod{m}$$

eine **eindeutige** Lösung: $a_j \equiv (x_j - y_j)^{-1} b \pmod{m}$

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

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

□

Familie für k -universelles Hashing

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

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

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

Satz

Für jede Primzahl m ist

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

eine k -universelle Familie von Hashfunktionen.

Familie für k -universelles Hashing

Beweisidee:

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

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

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

Falls $k \leq m$ können also höchstens $k-1$ von $m-1$ möglichen Werten für a zum gleichen Hashwert für \mathbf{x} und \mathbf{y} führen.

Aus $\Pr[h(\mathbf{x}) = h(\mathbf{y})] \leq \frac{\min\{k-1, m-1\}}{m-1} \leq \frac{k}{m}$ folgt, dass H' k -universell ist.

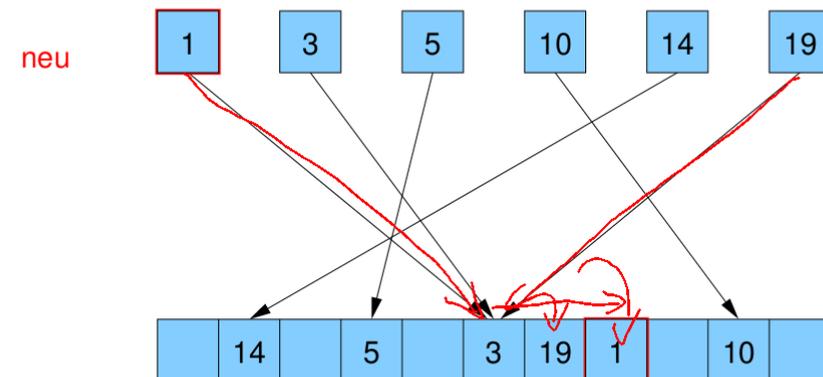
Übersicht

5 Hashing

- Hashtabellen
- Hashing with Chaining
- Universelles Hashing
- Hashing with Linear Probing
- Anpassung der Tabellengröße
- Perfektes Hashing
- Diskussion / Alternativen

Dynamisches Wörterbuch

Hashing with **Linear Probing**:



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

Hashing with Linear Probing

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

```
insert(Elem e) {
  i = h(key(e));
  while (T[i] ≠ null ∧ T[i] ≠ e)
    i = (i+1) % m;
  T[i] = e;
}
```

```
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.

⇒ Cache-Effizienz!

Hashing with Linear Probing

Problem: **Löschen** von Elementen

- 1 Löschen verbieten
 - 2 Markiere Positionen als gelöscht (mit speziellem Zeichen \perp)
Suche endet bei \perp , aber nicht bei markierten Zellen
- Problem: Anzahl echt freier Zellen sinkt monoton
⇒ Suche wird evt. langsam oder periodische Reorganisation

- 3 Invariante sicherstellen:

Für jedes $e \in S$ mit idealer Position $i = h(\text{key}(e))$ und aktueller Position j gilt:

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

Hashing with Linear Probing

Löschen / Aufrechterhaltung der Invariante

