

Script generated by TTT

Title: Seidl: GAD (03.05.2016)

Date: Tue May 03 14:21:28 CEST 2016

Duration: 90:09 min

Pages: 43

Datenstrukturen für Sequenzen Stacks und Queues

Beschränkte Queues

Methoden

```
void pushBack(Elem x) {  
    assert(size()<n);  
    b[t]=x;  
    t=(t+1)%(n+1);  
}  
  
void popFront() {  
    assert(!isEmpty());  
    h=(h+1)%(n+1);  
}  
  
int size() {  
    return (t-h+n+1)%(n+1);  
}
```

H. Seidl (TUM) GAD SS'16 141

Datenstrukturen für Sequenzen Stacks und Queues

Beschränkte Queues

- Struktur kann auch als **Deque** verwendet werden
- Zirkuläre Arrays erlauben auch den indexierten Zugriff:
Elem Operator [int i] {
 return b[(h+i)%(n+1)];
}
- Bounded Queues / Deques können genauso zu **Unbounded Queues / Deques** erweitert werden wie Bounded Arrays zu Unbounded Arrays

H. Seidl (TUM) GAD SS'16 142

Datenstrukturen für Sequenzen Diskussion: Sortierte Sequenzen

Sortierte Sequenz

S: sortierte Sequenz

Jedes Element e identifiziert über **key(e)**

Operationen:

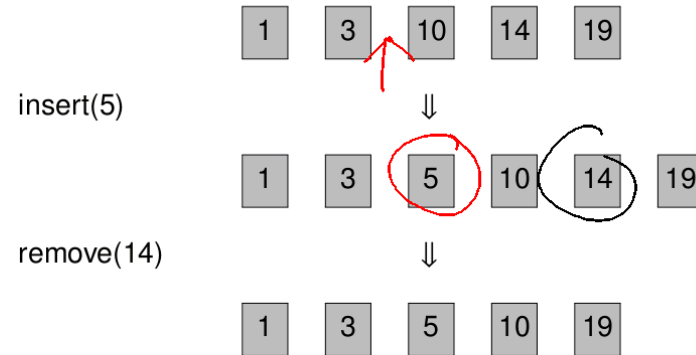
- $\langle e_1, \dots, e_n \rangle$.**insert**(e) = $\langle e_1, \dots, e_i, e, e_{i+1}, \dots, e_n \rangle$
für das i mit $\text{key}(e_i) < \text{key}(e) < \text{key}(e_{i+1})$
- $\langle e_1, \dots, e_n \rangle$.**remove**(k) = $\langle e_1, \dots, e_{i-1}, e_{i+1}, \dots, e_n \rangle$
für das i mit $\text{key}(e_i) = k$
- $\langle e_1, \dots, e_n \rangle$.**find**(k) = e_i
für das i mit $\text{key}(e_i) = k$

H. Seidl (TUM) GAD SS'16 143

Sortierte Sequenz

Problem:

Aufrechterhaltung der Sortierung nach jeder Einfügung / Löschung



Sortierte Sequenz

Realisierung als **Liste**

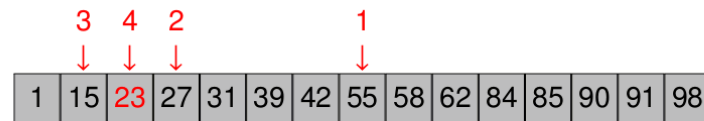
- insert und remove kosten zwar eigentlich nur konstante Zeit, müssen aber wie find zunächst die richtige Position finden
- find auf Sequenz der Länge n kostet $O(n)$ Zeit, damit ebenso insert und remove

Realisierung als **Feld**

- find kann mit binärer Suche in Zeit $O(\log n)$ realisiert werden
- insert und remove kosten $O(n)$ Zeit für das Verschieben der nachfolgenden Elemente

Binäre Suche

find(23):



In einer sortierten Sequenz mit n Elementen kann ein beliebiges Element mit $O(\log n)$ Vergleichen gefunden werden.

Assoziative Arrays / Wörterbücher

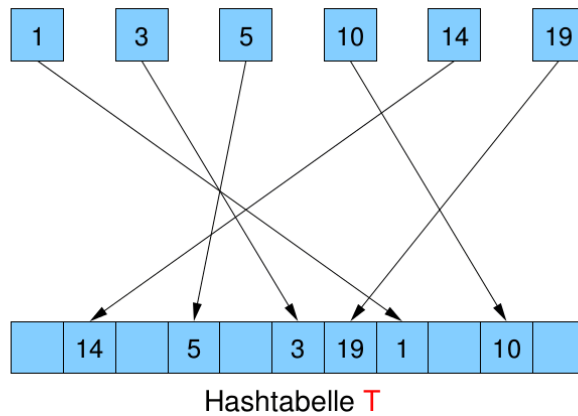
- Assoziatives Array / Wörterbuch (dictionary) **S**: speichert eine Menge von Elementen
- Element e wird identifiziert über eindeutigen Schlüssel **key**(e)

Operationen:

- **S.insert**(Elem e): $S := S \cup \{e\}$
- **S.remove**(Key k): $S := S \setminus \{e\}$, wobei e das Element mit $\text{key}(e) = k$ ist
- **S.find**(Key k): gibt das Element $e \in S$ mit $\text{key}(e) = k$ zurück, falls es existiert, sonst \perp (entspricht Array-Indexoperator $[]$, daher der Name)

Hashfunktion und Hashtabelle

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



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

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

```
void 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?

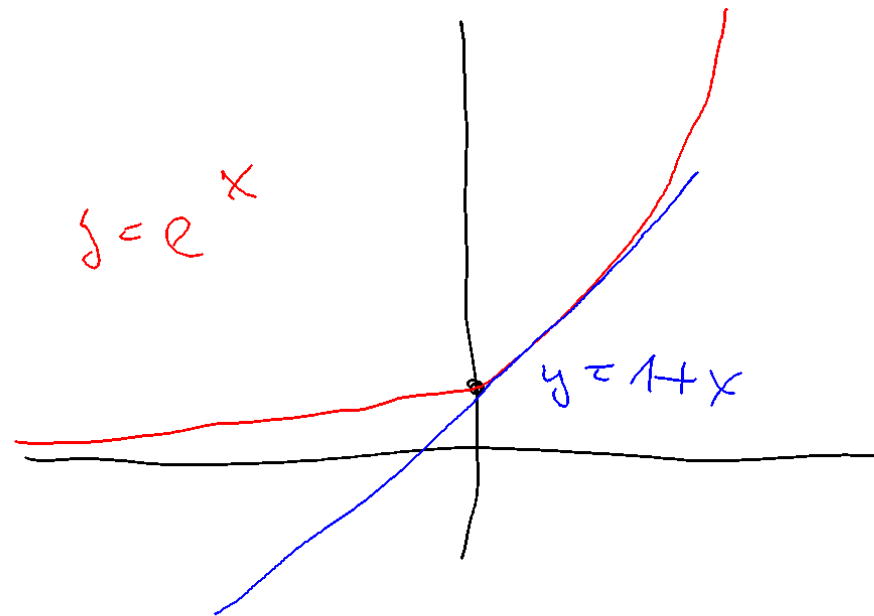
- Geburtstagsparadoxon: In einer Menge von 23 zufällig ausgewählten Personen gibt es mit Wahrscheinlichkeit $> 50\%$ zwei Leute, die am gleichen Tag Geburtstag feiern.
- Bei zufälliger Abbildung von 23 Schlüsseln auf die Adressen einer Hashtabelle der Größe 365 gibt es mit Wahrscheinlichkeit $> 50\%$ 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)
- $\Pr[\text{keine Kollision beim } i\text{-ten Schlüssel}] = \frac{m-(i-1)}{m}$ für $i \in [1 \dots n]$

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

- (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.



Wahrscheinlichkeit von Kollisionen

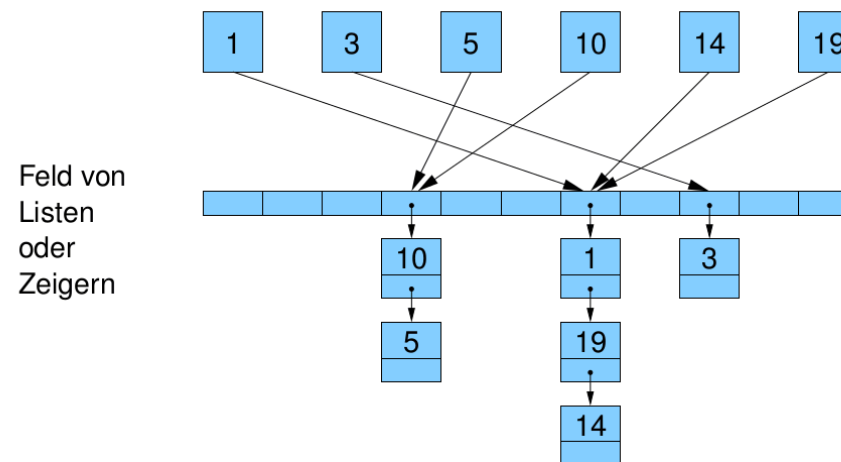
- Hilfsmittel: $\forall x \in \mathbb{R} : 1 + x \leq \sum_{i=0}^{\infty} \frac{x^i}{i!} = e^x$
- $\Rightarrow \forall x \in \mathbb{R} : \ln(1 + x) \leq x$ (da $\ln(x)$ monoton wachsend ist)
- $\Pr[\text{keine Kollision beim } i\text{-ten Schlüssel}] = \frac{m-(i-1)}{m}$ für $i \in [1 \dots n]$

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

- (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.

Dynamisches Wörterbuch

Hashing with **Chaining**:



Feld von Listen oder Zeigern

unsortierte verkettete Listen
(Ziel: Listen möglichst kurz)

Dynamisches Wörterbuch

Hashing with **Chaining**:

List<Elem>[m] T;

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

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

```
Elem find(Key k) {
    return 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 N/m mögliche Schlüssel zugeordnet sind (erweitertes **Schubfachprinzip** / pigeonhole principle)
- Meistens ist $n < N/m$ (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 $\log_2(m^N) = N \log_2 m$ Bits, um eine Funktion zu spezifizieren.

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

Hashing with Chaining

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 $\log_2(m^N) = N \log_2 m$ 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 `remove(k)` oder `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 `remove(k)` oder `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 ^{↑ + n/m}

Beweis.

- Betrachte feste Position $i = h(k)$ bei `remove(k)` oder `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 $\mathbb{E}[X]$

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

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 $\log_2(m^N) = N \log_2 m$ 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 `remove(k)` oder `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 $\mathbb{E}[X]$

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

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 $O(1 + c \cdot n/m)$.

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 $O(1 + c \cdot n/m)$.

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

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

□

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 $O(1 + c \cdot n/m)$.

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

Beispiele für c -universelles Hashing

Einfache c -universelle Hashfunktionen?

Annahme: Schlüssel sind Bitstrings einer bestimmten Länge

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

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

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

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

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

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

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

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:

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

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

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 $(x_j - y_j)^{-1}$
- ⇒ gegebene 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: $a_j \equiv (x_j - y_j)^{-1} b \pmod m$

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

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

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

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

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

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

□