

13. C++ vertieft (III): Funktoren und Lambda

Funktoren: Motivierung

Ein simpler Ausgabefilter

```
template <typename T, typename Function>
void filter(const T& collection, Function f){
    for (const auto& x: collection)
        if (f(x)) std::cout << x << " ";
    std::cout << "\n";
}
```

342

343

Funktoren: Motivierung

```
template <typename T, typename Function>
void filter(const T& collection, Function f);
```

```
template <typename T>
bool even(T x){
    %
    return x % 2 == 0;
}
```

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
filter(a,even<int>); // output: 2,4,6,16
```

344

Funktor: Objekt mit überladenem Operator ()

```
class GreaterThan{
    int value; // state
public:
    GreaterThan(int x):value{x}{}

    bool operator() (int par) const {
        return par > value;
    }
};
```

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int value=8;
filter(a,GreaterThan(value)); // 9,11,16,19
```

Funktor ist ein aufrufbares Objekt. Kann verstanden werden als Funktion mit Zustand.

345

Funktor: Objekt mit überladenem Operator ()

```
template <typename T>
class GreaterThan{
    T value;
public:
    GreaterThan(T x):value{x}{}

    bool operator() (T par) const{
        return par > value;
    }
};
```

(geht natürlich auch mit
Template)

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int value=8;
filter(a, GreaterThan<int>(value)); // 9,11,16,19
```

346

Dasselbe mit Lambda-Expression

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int value=8;

filter(a, [value](int x) {return x > value;});
```

347

Summe aller Elemente - klassisch

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int sum = 0;
for (auto x: a)
    sum += x;
std::cout << sum << "\n"; // 83
```

348

Summe aller Elemente - mit Funktor

```
template <typename T>
struct Sum{
    T value = 0;

    void operator() (T par){ value += par; }
};

std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
Sum<int> sum;
// for_each copies sum: we need to copy the result back
sum = std::for_each(a.begin(), a.end(), sum);
std::cout << sum.value << std::endl; // 83
```

349

Summe aller Elemente - mit Referenzen¹⁷

```
template <typename T>
struct SumR{
    T& value;
    SumR (T& v):value{v} {}

    void operator() (T par){ value += par; }
};

std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int s=0;
SumR<int> sum{s};
// cannot (and do not need to) assign to sum here
std::for_each(a.begin(), a.end(), sum);
std::cout << s << std::endl; // 83
```

¹⁷Geht natürlich sehr ähnlich auch mit Zeigern

350

Summe aller Elemente - mit Λ

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int s=0;

std::for_each(a.begin(), a.end(), [&s] (int x) {s += x;});

std::cout << s << "\n";
```

351

Sortieren, mal anders

```
// pre: i >= 0
// post: returns sum of digits of i
int q(int i){
    int res =0;
    for(;i>0;i/=10)
        res += i % 10;
    return res;
}

std::vector<int> v {10,12,9,7,28,22,14};
std::sort (v.begin(), v.end(),
    [] (int i, int j) { return q(i) < q(j);}
);
```

Jetzt $v = 10, 12, 22, 14, 7, 9, 28$ (sortiert nach Quersumme)

352

Lambda-Expressions im Detail

<u>[value]</u>	<u>(int x)</u>	<u>->bool</u>	<u>{return x > value;}</u>
Capture	Parameter	Rückgabe- typ	Anweisung

353

Closure

```
[value] (int x) ->bool {return x > value;}
```

- Lambda-Expressions evaluieren zu einem temporären Objekt – einer closure
- Die closure erhält den Ausführungskontext der Funktion, also die captured Objekte.
- Lambda-Expressions können als Funktoren implementiert werden.

354

Simple Lambda-Expression

```
[] () ->void {std::cout << "Hello World";}
```

Aufruf:

```
[] () ->void {std::cout << "Hello World";}();
```

355

Minimale Lambda-Expression

```
[] {}
```

- Rückgabtyp kann inferiert werden, wenn kein oder nur ein return:¹⁸

```
[] () {std::cout << "Hello World";}
```
- Keine Parameter und kein expliziter Rückgabtyp \Rightarrow () kann weggelassen werden

```
[] {std::cout << "Hello World";}
```
- [...] kann nie weggelassen werden.

¹⁸Seit C++14 auch mehrere returns, sofern derselbe Rückgabtyp deduziert wird

356

Beispiele

```
[] (int x, int y) {std::cout << x * y;} (4,5);
```

Output: 20

357

Beispiele

```
int k = 8;
[](int& v) {v += v;} (k);
std::cout << k;
```

Output: 16

358

Beispiele

```
int k = 8;
[](int v) {v += v;} (k);
std::cout << k;
```

Output: 8

359

Capture – Lambdas

Für Lambda-Expressions bestimmt die capture-Liste über den zugreifbaren Teil des Kontextes

Syntax:

- `[x]`: Zugriff auf kopierten Wert von x (nur lesend)
- `&x`: Zugriff zur Referenz von x
- `&x, y`: Zugriff zur Referenz von x und zum kopierten Wert von y
- `&`: Default-Referenz-Zugriff auf alle Objekte im Scope der Lambda-Expression
- `=`: Default-Werte-Zugriff auf alle Objekte im Kontext der Lambda-Expression

360

Capture – Lambdas

```
int elements=0;
int sum=0;
std::for_each(v.begin(), v.end(),
    [&] (int k) {sum += k; elements++;} // capture all by reference
)
```

361

Capture – Lambdas

```
template <typename T>
void sequence(vector<int> & v, T done){
    int i=0;
    while (!done()) v.push_back(i++);
}

vector<int> s;
sequence(s, [&] {return s.size() >= 5; } )
```

jetzt v = 0 1 2 3 4

Die capture liste bezieht sich auf den Kontext der Lambda Expression

362

Capture – Lambdas

Wann wird der Wert gelesen?

```
int v = 42;
auto func = [=] {std::cout << v << "\n"};
v = 7;
func();
```

Ausgabe: 42

Werte werden bei der Definition der (temporären) Lambda-Expression zugewiesen.

363

Capture – Lambdas

(Warum) funktioniert das?

```
class Limited{
    int limit = 10;
public:
    // count entries smaller than limit
    int count(const std::vector<int>& a){
        int c = 0;
        std::for_each(a.begin(), a.end(),
            [=,&c] (int x) {if (x < limit) c++;}
        );
        return c;
    }
};
```

Der **this** pointer wird per default implizit kopiert

364

Capture – Lambdas

```
struct mutant{
    int i = 0;
    void do(){ [=] {i=42;}();}
};
```

```
mutant m;
m.do();
std::cout << m.i;
```

Ausgabe: 42

Der **this pointer** wird per default implizit kopiert

365

Lambda Ausdrücke sind Funktoren

```
[x, &y] () {y = x;}
```

kann implementiert werden als

```
unnamed {x,y};
```

mit

```
class unnamed {  
    int x; int& y;  
    unnamed (int x_, int& y_) : x (x_), y (y_) {}  
    void operator () () {y = x;}  
};
```

366

Lambda Ausdrücke sind Funktoren

```
[=] () {return x + y;}
```

kann implementiert werden als

```
unnamed {x,y};
```

mit

```
class unnamed {  
    int x; int y;  
    unnamed (int x_, int y_) : x (x_), y (y_) {}  
    int operator () () const {return x + y;}  
};
```

367

Polymorphic Function Wrapper `std::function`

```
#include <functional>
```

```
int k= 8;  
std::function<int(int)> f;  
f = [k](int i){ return i+k; };  
std::cout << f(8); // 16
```

Kann verwendet werden, um Lambda-Expressions zu speichern.

Andere Beispiele

```
std::function<int(int,int)>;  
std::function<void(double)> ...
```

<http://en.cppreference.com/w/cpp/utility/functional/function>

368

14. Hashing

Hash Tabellen, Geburtstagsparadoxon, Hashfunktionen, Perfektes und universelles Hashing, Kollisionsauflösung durch Verkettung, offenes Hashing, Sondieren [Ottman/Widmayer, Kap. 4.1-4.3.2, 4.3.4, Cormen et al, Kap. 11-11.4]

369

Motivation

Ziel: Tabelle aller n Studenten dieser Vorlesung

Anforderung: Schneller Zugriff per Name

Naive Ideen

Zuordnung Name $s = s_1s_2 \dots s_{l_s}$ zu Schlüssel

$$k(s) = \sum_{i=1}^{l_s} s_i \cdot b^i$$

b gross genug, so dass verschiedene Namen verschiedene Schlüssel erhalten.

Speichere jeden Datensatz an seinem Index in einem grossen Array.

Beispiel, mit $b = 100$. Ascii-Werte s_i .

Anna \mapsto 71111065

Jacqueline \mapsto 102110609021813999774

Unrealistisch: erfordert zu grosse Arrays.

370

371

Bessere Idee?

Allokation eines Arrays der Länge m ($m > n$).

Zuordnung Name s zu

$$k_m(s) = \left(\sum_{i=1}^{l_s} s_i \cdot b^i \right) \bmod m.$$

Verschiedene Namen können nun denselben Schlüssel erhalten ("Kollision"). Und dann?

Abschätzung

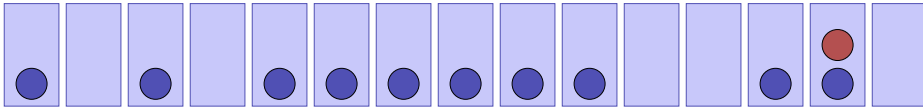
Vielleicht passieren Kollisionen ja fast nie. Wir schätzen ab ...

372

373

Abschätzung

Annahme: m Urnen, n Kugeln (oBdA $n \leq m$).
 n Kugeln werden gleichverteilt in Urnen gelegt.



Wie gross ist die Kollisionswahrscheinlichkeit?

Sehr verwandte Frage: Bei wie vielen Personen (n) ist die Wahrscheinlichkeit, dass zwei am selben Tag ($m = 365$) Geburtstag haben grösser als 50%?

Abschätzung

$$\mathbb{P}(\text{keine Kollision}) = \frac{m}{m} \cdot \frac{m-1}{m} \cdot \dots \cdot \frac{m-n+1}{m} = \frac{m!}{(m-n)! \cdot m^n}.$$

Sei $a \ll m$. Mit $e^x = 1 + x + \frac{x^2}{2!} + \dots$ approximiere $1 - \frac{a}{m} \approx e^{-\frac{a}{m}}$.

Damit:

$$1 \cdot \left(1 - \frac{1}{m}\right) \cdot \left(1 - \frac{2}{m}\right) \cdot \dots \cdot \left(1 - \frac{n-1}{m}\right) \approx e^{-\frac{1+\dots+n-1}{m}} = e^{-\frac{n(n-1)}{2m}}.$$

Es ergibt sich

$$\mathbb{P}(\text{Kollision}) = 1 - e^{-\frac{n(n-1)}{2m}}.$$

Auflösung zum Geburtstagsparadoxon: Bei 23 Leuten ist die Wahrscheinlichkeit für Geburtstagskollision 50.7%. Zahl stammt von der leicht besseren Approximation via Stirling Formel.

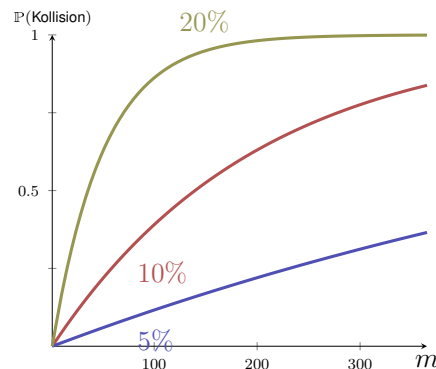
374

375

Mit Füllgrad

Mit Füllgrad $\alpha := n/m$ ergibt sich (weiter vereinfacht)

$$\mathbb{P}(\text{Kollision}) \approx 1 - e^{-\alpha^2 \cdot \frac{m}{2}}.$$



Andere Frage

Annahme: m Urnen, n Kugeln (oBdA $n \leq m$).
 n Kugeln werden gleichverteilt in Urnen gelegt.



Wie gross ist die erwartete Anzahl von Kollisionen?

376

377

Erwartete Anzahl Kollisionen

- $\mathbb{P}(\text{Kugel } B \text{ trifft Kugel } A_i) = 1/m.$
- $\mathbb{P}(\text{Kugel } B \text{ trifft Kugel } A_i \text{ nicht}) = 1 - 1/m.$
- $\mathbb{P}(n - 1 \text{ Kugeln treffen } A_i \text{ nicht}) = (1 - 1/m)^{n-1}.$
- $\mathbb{P}(A_i \text{ getroffen}) = 1 - (1 - 1/m)^{n-1}.$
- Sei X_i Zufallsvariable mit $X_i = \mathbb{1}_{A_i \text{ getroffen}}$
- $\mathbb{E}(\sum X_i) = \sum \mathbb{E}(X_i)$
- $\mathbb{E}(\text{Anzahl getroffene Kugeln}) = n(1 - (1 - 1/m)^n) \approx \frac{n^2}{2m}.$

378

Nomenklatur

Hashfunktion h : Abbildung aus der Menge der Schlüssel \mathcal{K} auf die Indexmenge $\{0, 1, \dots, m - 1\}$ eines Arrays (**Hashtabelle**).

$$h : \mathcal{K} \rightarrow \{0, 1, \dots, m - 1\}.$$

Meist $|\mathcal{K}| \gg m$. Es gibt also $k_1, k_2 \in \mathcal{K}$ mit $h(k_1) = h(k_2)$ (**Kollision**).

Eine Hashfunktion sollte die Menge der Schlüssel möglichst gleichmässig auf die Positionen der Hashtabelle verteilen.

379

Beispiele guter Hashfunktionen

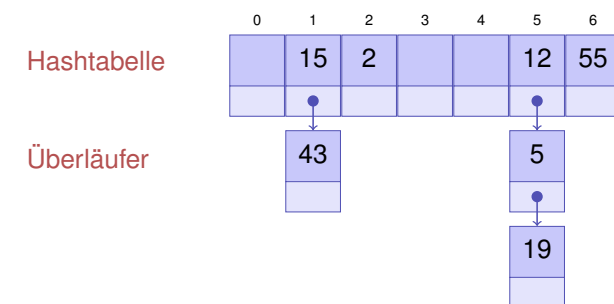
- $h(k) = k \bmod m$, m Primzahl
- $h(k) = \lfloor m(k \cdot r - \lfloor k \cdot r \rfloor) \rfloor$, r irrational, besonders gut: $r = \frac{\sqrt{5}-1}{2}$.

380

Behandlung von Kollisionen

Beispiel $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod m$.
Schlüssel 12, 55, 5, 15, 2, 19, 43

Verkettung der Überläufer



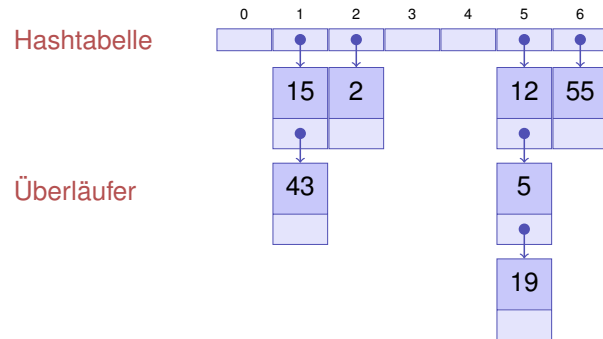
381

Behandlung von Kollisionen

Beispiel $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod m$.

Schlüssel 12, 55, 5, 15, 2, 19, 43

Direkte Verkettung der Überläufer



382

Algorithmen zum Hashing mit Verkettung

- **contains**(k) Durchsuche Liste an Position $h(k)$ nach k . Gib wahr zurück, wenn gefunden, sonst falsch.
- **put**(k) Prüfe ob k in Liste an Position $h(k)$. Falls nein, füge k am Ende der Liste ein. Andernfalls Fehlermeldung.
- **get**(k) Prüfe ob k in Liste an Position $h(k)$. Falls ja, gib die Daten zum Schlüssel k zurück. Andernfalls Fehlermeldung
- **remove**(k) Durchsuche die Liste an Position $h(k)$ nach k . Wenn Suche erfolgreich, entferne das entsprechende Listenelement.

383

Analyse (direkt verkettete Liste)

- 1 Erfolgreiche Suche. Durchschnittliche Listenlänge ist $\alpha = \frac{n}{m}$. Liste muss ganz durchlaufen werden.

⇒ Durchschnittliche Anzahl betrachteter Einträge

$$C'_n = \alpha.$$

- 2 Erfolgreiche Suche. Betrachten die Einfügehistorie: Schlüssel j sieht durchschnittliche Listenlänge $(j-1)/m$.

⇒ Durchschnittliche Anzahl betrachteter Einträge

$$C_n = \frac{1}{n} \sum_{j=1}^n (1 + (j-1)/m) = 1 + \frac{1}{n} \frac{n(n-1)}{2m} \approx 1 + \frac{\alpha}{2}.$$

384

Vor und Nachteile

Vorteile der Strategie:

- Belegungsfaktoren $\alpha > 1$ möglich
- Entfernen von Schlüsseln einfach

Nachteile

- Speicherverbrauch der Verkettung

385

Offene Hashverfahren

Speichere die Überläufer direkt in der Hashtabelle mit einer **Sondierfunktion** $s(j, k)$ ($0 \leq j < m, k \in \mathcal{K}$)

Tabellenposition des Schlüssels entlang der **Sondierungsfolge**

$$S(k) := ((h(k) + s(0, k)) \bmod m, \dots, (h(k) + s(m - 1, k)) \bmod m)$$

386

Algorithmen zum Open Addressing

- **contains**(k) Durchlaufe Tabelleneinträge gemäss $S(k)$. Wird k gefunden, gib **true** zurück. Ist die Sondierungsfolge zu Ende oder eine leere Position erreicht, gib **false** zurück.
- **put**(k) Suche k in der Tabelle gemäss $S(k)$. Ist k nicht vorhanden, füge k an die erste freie Position in der Sondierungsfolge ein. Andernfalls Fehlermeldung.
- **get**(k) Durchlaufe Tabelleneinträge gemäss $S(k)$. Wird k gefunden, gib die zu k gehörenden Daten zurück. Andernfalls Fehlermeldung.
- **remove**(k) Suche k in der Tabelle gemäss $S(k)$. Wenn k gefunden, ersetze k durch den speziellen **removed** key.

387

Lineares Sondieren

$$s(j, k) = j \Rightarrow S(k) = (h(k) \bmod m, (h(k) + 1) \bmod m, \dots, (h(k) - 1) \bmod m)$$

Beispiel $m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \bmod m$.
Schlüssel 12, 55, 5, 15, 2, 19

0	1	2	3	4	5	6
5	15	2	19		12	55

388

Analyse Lineares Sondieren (ohne Herleitung)

- 1 Erfolgreiche Suche. Durchschnittliche Anzahl betrachteter Einträge

$$C'_n \approx \frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2} \right)$$

- 2 Erfolgreiche Suche. Durchschnittliche Anzahl betrachteter Einträge

$$C_n \approx \frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right).$$

389

Diskussion

Beispiel $\alpha = 0.95$

Erfolgreiche Suche betrachtet im Durchschnitt 200 Tabelleneinträge!

❓ Grund für die schlechte Performance?

⚠️ **Primäre Häufung:** Ähnliche Hashadressen haben ähnliche Sondierungsfolgen \Rightarrow lange zusammenhängende belegte Bereiche.

Quadratisches Sondieren

$$s(j, k) = \lceil j/2 \rceil^2 (-1)^{j+1}$$

$$S(k) = (h(k), h(k) + 1, h(k) - 1, h(k) + 4, h(k) - 4, \dots) \pmod m$$

Beispiel $m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod m$.

Schlüssel 12, 55, 5, 15, 2, 19

0	1	2	3	4	5	6
19	15	2		5	12	55

390

391

Analyse Quadratisches Sondieren (ohne Herleitung)

- 1 Erfolgreiche Suche. Durchschnittliche Anzahl betrachteter Einträge

$$C'_n \approx \frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{1-\alpha}\right)$$

- 2 Erfolgreiche Suche. Durchschnittliche Anzahl betrachteter Einträge

$$C_n \approx 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2}$$

Diskussion

Beispiel $\alpha = 0.95$

Erfolgreiche Suche betrachtet im Durchschnitt 22 Tabelleneinträge

❓ Grund für die schlechte Performance?

⚠️ **Sekundäre Häufung:** Synonyme k und k' (mit $h(k) = h(k')$) durchlaufen dieselbe Sondierungsfolge.

392

393

Double Hashing

Zwei Hashfunktionen $h(k)$ und $h'(k)$. $s(j, k) = j \cdot h'(k)$.
 $S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \dots, h(k) + (m-1)h'(k)) \pmod m$

Beispiel:
 $m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod 7, h'(k) = 1 + k \pmod 5$.
 Schlüssel 12, 55, 5, 15, 2, 19

0	1	2	3	4	5	6
5	15	2	19		12	55

394

Double Hashing

- Sondierungsreihenfolge muss Permutation aller Hashadressen bilden. Also $h'(k) \neq 0$ und $h'(k)$ darf m nicht teilen, z.B. garantiert mit m prim.
- h' sollte unabhängig von h sein (Vermeidung sekundärer Häufung).

Unabhängigkeit:

$$\mathbb{P}((h(k) = h(k')) \wedge (h'(k) = h'(k'))) = \mathbb{P}(h(k) = h(k')) \cdot \mathbb{P}(h'(k) = h'(k')).$$

Unabhängigkeit erfüllt von $h(k) = k \pmod m$ und $h'(k) = 1 + k \pmod (m-2)$ (m prim).

395

Analyse Double Hashing

Sind h und h' unabhängig, dann:

- 1 Erfolgreiche Suche. Durchschnittliche Anzahl betrachteter Einträge

$$C'_n \approx \frac{1}{1 - \alpha}$$

- 2 Erfolgreiche Suche. Durchschnittliche Anzahl betrachteter Einträge

$$C_n \approx \frac{1}{\alpha} \ln \left(\frac{1}{1 - \alpha} \right)$$

396

Übersicht

	$\alpha = 0.50$		$\alpha = 0.90$		$\alpha = 0.95$	
	C_n	C'_n	C_n	C'_n	C_n	C'_n
Separate Verkettung	1.250	1.110	1.450	1.307	1.475	1.337
Direkte Verkettung	1.250	0.500	1.450	0.900	1.475	0.950
Lineares Sondieren	1.500	2.500	5.500	50.500	10.500	200.500
Quadratisches Sondieren	1.440	2.190	2.850	11.400	3.520	22.050
Double Hashing	1.39	2.000	2.560	10.000	3.150	20.000

: C_n : Anzahl Schritte erfolgreiche Suche, C'_n : Anzahl Schritte erfolglose Suche, Belegungsgrad α .

397

Perfektes Hashing

Ist im Vorhinein die Menge der verwendeten Schlüssel bekannt? Dann kann die Hashfunktion perfekt, also kollisionsfrei, gewählt werden. Die praktische Konstruktion ist nicht-trivial.

Beispiel: Tabelle der Schlüsselwörter in einem Compiler.

Universelles Hashing

- $|\mathcal{K}| > m \Rightarrow$ Menge “ähnlicher Schlüssel” kann immer so gewählt sein, so dass überdurchschnittlich viele Kollisionen entstehen.
- Unmöglich, einzelne für alle Fälle “beste” Hashfunktion auszuwählen.
- Jedoch möglich¹⁹: randomisieren!

Universelle Hashklasse $\mathcal{H} \subseteq \{h : \mathcal{K} \rightarrow \{0, 1, \dots, m-1\}\}$ ist eine Familie von Hashfunktionen, so dass

$$\forall k_1 \neq k_2 \in \mathcal{K} : |\{h \in \mathcal{H} | h(k_1) = h(k_2)\}| \leq \frac{1}{m} |\mathcal{H}|.$$

¹⁹Ähnlich wie beim Quicksort

Universelles Hashing

Theorem

Eine aus einer universellen Klasse \mathcal{H} von Hashfunktionen zufällig gewählte Funktion $h \in \mathcal{H}$ verteilt im Erwartungswert eine beliebige Folge von Schlüssel aus \mathcal{K} so gleichmässig wie nur möglich auf die verfügbaren Plätze.

Universelles Hashing

Vorbemerkung zum Beweis des Theorems.

Definiere mit $x, y \in \mathcal{K}$, $h \in \mathcal{H}$, $Y \subseteq \mathcal{K}$:

$$\delta(x, y, h) = \begin{cases} 1, & \text{falls } h(x) = h(y), x \neq y \\ 0, & \text{sonst,} \end{cases}$$

$$\delta(x, Y, h) = \sum_{y \in Y} \delta(x, y, h),$$

$$\delta(x, y, \mathcal{H}) = \sum_{h \in \mathcal{H}} \delta(x, y, h).$$

\mathcal{H} ist universell, wenn für alle $x, y \in \mathcal{K}$, $x \neq y$: $\delta(x, y, \mathcal{H}) \leq |\mathcal{H}|/m$.

Universelles Hashing

Beweis des Theorems

$S \subseteq \mathcal{K}$: bereits gespeicherte Schlüssel. x wird hinzugefügt:

$$\begin{aligned}\mathbb{E}_{\mathcal{H}}(\delta(x, S, h)) &= \sum_{h \in \mathcal{H}} \delta(x, S, h) / |\mathcal{H}| \\ &= \frac{1}{|\mathcal{H}|} \sum_{h \in \mathcal{H}} \sum_{y \in S} \delta(x, y, h) = \frac{1}{|\mathcal{H}|} \sum_{y \in S} \sum_{h \in \mathcal{H}} \delta(x, y, h) \\ &= \frac{1}{|\mathcal{H}|} \sum_{y \in S} \delta(x, y, \mathcal{H}) \\ &\leq \frac{1}{|\mathcal{H}|} \sum_{y \in S} |\mathcal{H}| / m = \frac{|S|}{m}.\end{aligned}$$

■

402

Universelles Hashing ist relevant!

Sei p Primzahl und $\mathcal{K} = \{0, \dots, p-1\}$. Mit $a \in \mathcal{K} \setminus \{0\}$, $b \in \mathcal{K}$ definiere

$$h_{ab} : \mathcal{K} \rightarrow \{0, \dots, m-1\}, h_{ab}(x) = ((ax + b) \bmod p) \bmod m.$$

Dann gilt

Theorem

Die Klasse $\mathcal{H} = \{h_{ab} \mid a, b \in \mathcal{K}, a \neq 0\}$ ist eine universelle Klasse von Hashfunktionen.

403