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";
}
```

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
```

Funktor: Objekt mit überladenem Operator ()

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

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

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

```
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: 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
```

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;} );
```

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
```

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
```

Summe aller Elemente - mit Referenzen[17]

```
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
```

[17] Geht natürlich sehr ähnlich auch mit Zeigern

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";
```

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);}
);
```

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 $v =$

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)

Lambda-Expressions im Detail

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

Capture — Parameter — Rück-gabe-typ — Anweisung

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.

Simple Lambda-Expression

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

Simple Lambda-Expression

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

Aufruf:

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

Minimale Lambda-Expression

```
[]{}
```

- Rückgabetyp kann inferiert werden, wenn kein oder nur ein return:[18]

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

- Keine Parameter und kein expliziter Rückgabetyp ⇒ () kann weggelassen werden

```
[]{std::cout << "Hello World";}
```

- `[...]` kann nie weggelassen werden.

[18] Seit C++14 auch mehrere returns, sofern derselbe Rückgabetyp deduziert wird

Beispiele

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

Output:

Beispiele

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

Output: 20

Beispiele

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

Output:

Beispiele

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

Output: 16

Beispiele

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

Output:

Beispiele

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

Output: 8

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

Capture – Lambdas

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

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 =

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

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

Capture – Lambdas

Wann wird der Wert gelesen?

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

Ausgabe:

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.

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;
  }
};
```

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

Capture – Lambdas

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

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

Ausgabe:

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

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;}
};
```

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;}
};
```

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

14. Hashing

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

Motivation

Ziel: Tabelle aller n Studenten dieser Vorlesung

Anforderung: Schneller Zugriff per Name

Naive Ideen

Zuordnung Name $s = s_1 s_2 \dots s_{l_s}$ zu Schlüssel

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

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

Naive Ideen

Zuordnung Name $s = s_1 s_2 \dots s_{l_s}$ 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.

Naive Ideen

Zuordnung Name $s = s_1 s_2 \dots s_{l_s}$ 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 $\mapsto 102110609021813999774$

Naive Ideen

Zuordnung Name $s = s_1 s_2 \dots s_{l_s}$ 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.

Bessere Idee?

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

Bessere Idee?

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

Zuordnung Name s zu

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

Bessere Idee?

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

Zuordnung Name s zu

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

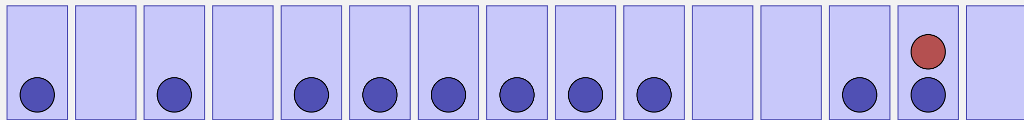
Verschiedene Namen können nun denselben Schlüssel erhalten (“Kollision”). Und dann?

Abschätzung

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

Abschätzung

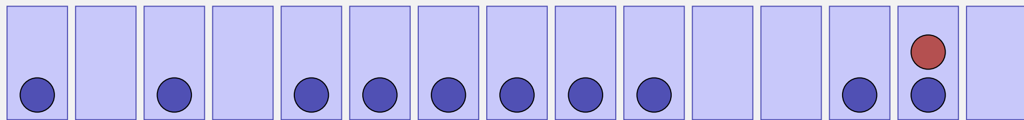
Annahme: m Urnen, n Kugeln (oBdA $n \leq m$).
n Kugeln werden gleichverteilt in Urnen gelegt.



Wie gross ist die Kollisionswahrscheinlichkeit?

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

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

Abschätzung

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

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

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

Abschätzung

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

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:

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

Es ergibt sich

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

Abschätzung

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

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

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

Es ergibt sich

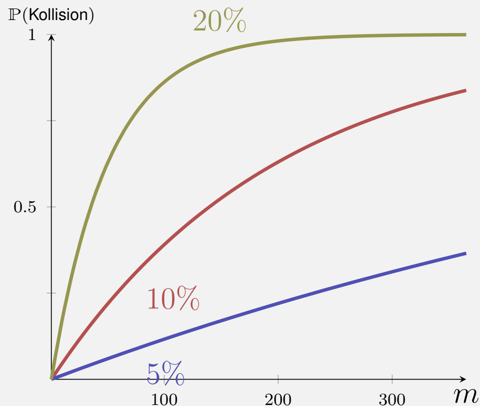
$$\mathbb{P}(\text{Kollision}) = 1 - e^{-\frac{n(n-1)}{2m}}.$$

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

Mit Füllgrad

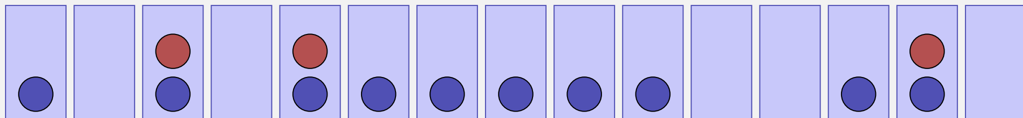
Mit Füllgrad $\alpha := n/m$ ergibt sich (weiter vereinfacht)

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



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?

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 $X_i = \mathbb{1}_{A_i \text{getroffen}}$
- $\mathbb{E}(\sum X_i) = \sum \mathbb{E}(X_i)$
- $\mathbb{E}(\text{Anzahl getroffene Kugeln}) = n(1 - (1 - 1/m)^n) \approx \frac{n^2}{2m}$.

Nomenklatur

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

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

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.

Beispiele guter Hashfunktionen

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

Behandlung von Kollisionen

Beispiel $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$. Schlüssel $12, 53, 15, 2, 19, 43$

Behandlung von Kollisionen

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

Schlüssel 12 , 55

Verkettung der Überläufer

Hashtabelle

0	1	2	3	4	5	6
					12	

Überläufer

Behandlung von Kollisionen

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

Schlüssel 12 , 55 , 5

Verkettung der Überläufer

Hashtabelle

0	1	2	3	4	5	6
					12	55

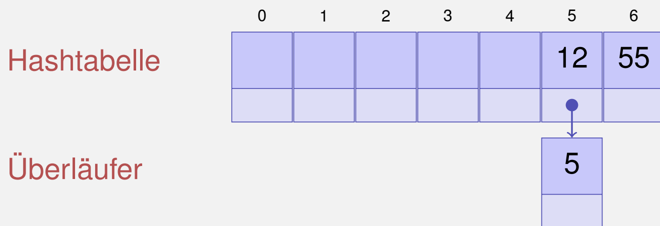
Überläufer

Behandlung von Kollisionen

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

Schlüssel 12 , 55 , 5 , 15

Verkettung der Überläufer

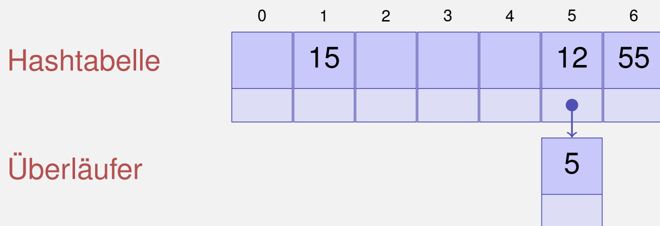


Behandlung von Kollisionen

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

Schlüssel 12 , 55 , 5 , 15 , 2

Verkettung der Überläufer

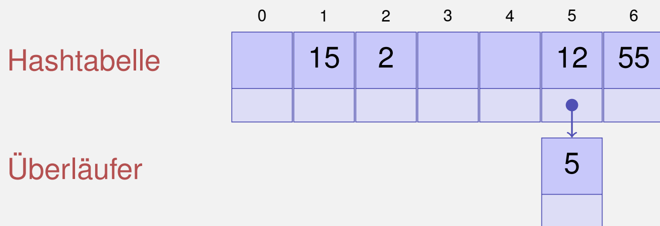


Behandlung von Kollisionen

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

Schlüssel 12 , 55 , 5 , 15 , 2 , 19

Verkettung der Überläufer

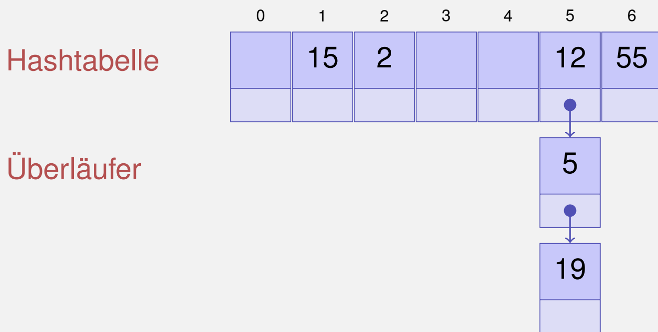


Behandlung von Kollisionen

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

Schlüssel 12 , 55 , 5 , 15 , 2 , 19 , 43

Verkettung der Überläufer

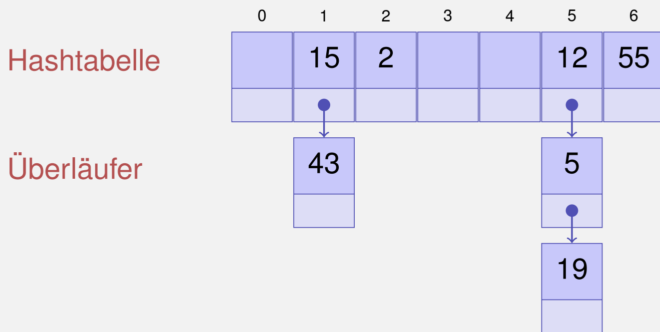


Behandlung von Kollisionen

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

Schlüssel 12 , 55 , 5 , 15 , 2 , 19 , 43

Verkettung der Überläufer



Behandlung von Kollisionen

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

Schlüssel 12

Direkte Verkettung der Überläufer

Hashtabelle

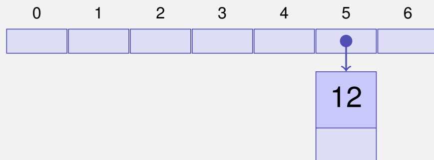
Überläufer

Behandlung von Kollisionen

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

Schlüssel 12 , 55

Direkte Verkettung der Überläufer



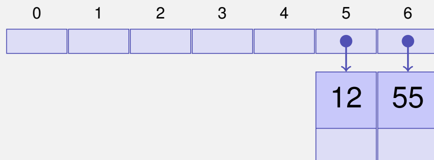
Behandlung von Kollisionen

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

Schlüssel 12 , 55 , 5

Direkte Verkettung der Überläufer

Hashtabelle



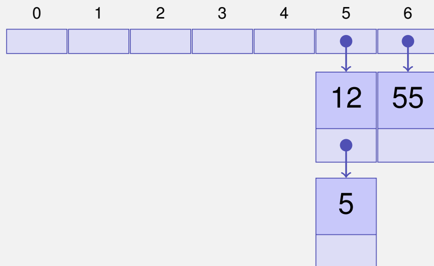
Überläufer

Behandlung von Kollisionen

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

Schlüssel 12 , 55 , 5 , 15

Direkte Verkettung der Überläufer

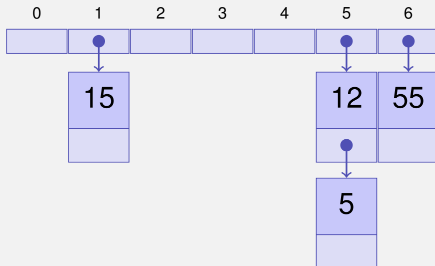


Behandlung von Kollisionen

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

Schlüssel 12 , 55 , 5 , 15 , 2

Direkte Verkettung der Überläufer

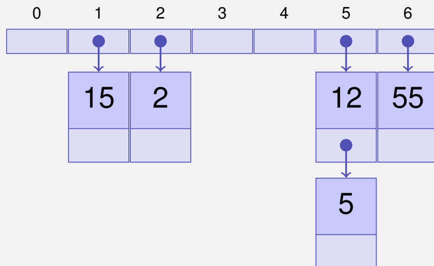


Behandlung von Kollisionen

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

Schlüssel 12 , 55 , 5 , 15 , 2 , 19

Direkte Verkettung der Überläufer



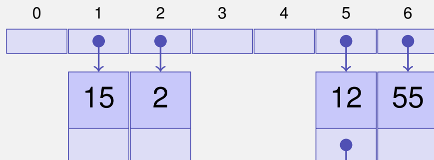
Behandlung von Kollisionen

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

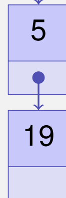
Schlüssel 12 , 55 , 5 , 15 , 2 , 19 , 43

Direkte Verkettung der Überläufer

Hashtabelle



Überläufer

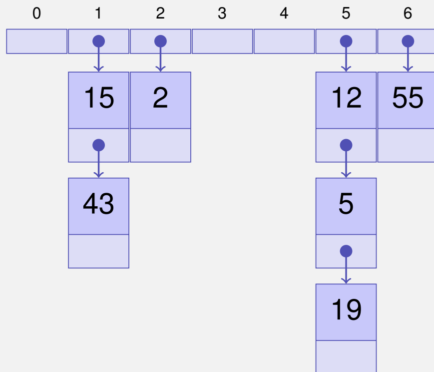


Behandlung von Kollisionen

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

Schlüssel 12 , 55 , 5 , 15 , 2 , 19 , 43

Direkte Verkettung der Überläufer



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.

Analyse (direkt verkettete Liste)

1. Erfolglose Suche.

Analyse (direkt verkettete Liste)

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

Analyse (direkt verkettete Liste)

1. Erfolglose Suche. Durchschnittliche Listenlänge ist $\alpha = \frac{n}{m}$. Liste muss ganz durchlaufen werden.
   $\Rightarrow$ Durchschnittliche Anzahl betrachteter Einträge

$$C'_n = \alpha.$$

Analyse (direkt verkettete Liste)

1. Erfolglose Suche. Durchschnittliche Listenlänge ist $\alpha = \frac{n}{m}$. Liste muss ganz durchlaufen werden.
   $\Rightarrow$ Durchschnittliche Anzahl betrachteter Einträge

$$C'_n = \alpha.$$

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

Analyse (direkt verkettete Liste)

1. Erfolglose Suche. Durchschnittliche Listenlänge ist $\alpha = \frac{n}{m}$. Liste muss ganz durchlaufen werden.
   $\Rightarrow$ Durchschnittliche Anzahl betrachteter Einträge

$$C'_n = \alpha.$$

2. Erfolgreiche Suche. Betrachten die Einfügehistorie: Schlüssel j sieht durchschnittliche Listenlänge $(j-1)/m$.
   $\Rightarrow$ Durchschnittliche Anzahl betrachteter Einträge

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

Analyse (direkt verkettete Liste)

1. Erfolglose Suche. Durchschnittliche Listenlänge ist $\alpha = \frac{n}{m}$. Liste muss ganz durchlaufen werden.
   $\Rightarrow$ Durchschnittliche Anzahl betrachteter Einträge

$$C'_n = \alpha.$$

2. Erfolgreiche Suche. Betrachten die Einfügehistorie: Schlüssel j sieht durchschnittliche Listenlänge $(j-1)/m$.
   $\Rightarrow$ Durchschnittliche Anzahl betrachteter Einträge

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

Analyse (direkt verkettete Liste)

1. Erfolglose Suche. Durchschnittliche Listenlänge ist $\alpha = \frac{n}{m}$. Liste muss ganz durchlaufen werden.
   $\Rightarrow$ Durchschnittliche Anzahl betrachteter Einträge
   $$C'_n = \alpha.$$

2. Erfolgreiche Suche. Betrachten die Einfügehistorie: Schlüssel j sieht durchschnittliche Listenlänge $(j-1)/m$.
   $\Rightarrow$ Durchschnittliche Anzahl betrachteter Einträge
   $$C_n = \frac{1}{n}\sum_{j=1}^{n}(1 + (j-1)/m)) = 1 + \frac{1}{n}\frac{n(n-1)}{2m} \approx 1 + \frac{\alpha}{2}.$$

Vor und Nachteile

Vorteile der Strategie:

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

Nachteile

- Speicherverbrauch der Verkettung

Offene Hashverfahren

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

Tabellenposition des Schlüssels entlang der *Sondierungsfolge*

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

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.

Lineares Sondieren

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

Lineares Sondieren

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

Beispiel $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.
Schlüssel 12

0	1	2	3	4	5	6

Lineares Sondieren

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

Beispiel $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.
Schlüssel 12 , 55

0	1	2	3	4	5	6
					12	

Lineares Sondieren

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

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

0	1	2	3	4	5	6
					12	55

Lineares Sondieren

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

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

0	1	2	3	4	5	6
5					12	55

Lineares Sondieren

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

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

0	1	2	3	4	5	6
5	15				12	55

Lineares Sondieren

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

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

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

Lineares Sondieren

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

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

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

Analyse Lineares Sondieren (ohne Herleitung)

Analyse Lineares Sondieren (ohne Herleitung)

1. Erfolglose Suche. Durchschnittliche Anzahl betrachteter Einträge

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

Analyse Lineares Sondieren (ohne Herleitung)

1. Erfolglose 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

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

Diskussion

Diskussion

Beispiel $\alpha = 0.95$

Erfolglose Suche betrachtet im Durchschnitt 200 Tabelleneinträge!

Diskussion

Beispiel $\alpha = 0.95$

Erfolglose Suche betrachtet im Durchschnitt 200 Tabelleneinträge!

? Grund für die schlechte Performance?

Diskussion

Beispiel $\alpha = 0.95$

Erfolglose Suche betrachtet im Durchschnitt 200 Tabelleneinträge!

? Grund für die schlechte Performance?

! *Primäre Häufung:* Ähnliche Hashaddressen haben ähnliche Sondierungsfolgen $\Rightarrow$ lange zusammenhängende belegte Bereiche.

Quadratisches Sondieren

$$s(j,k) = \lceil j/2 \rceil^2 (-1)^{j+1}$$
$$S(k) = (h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \dots) \mod m$$

Quadratisches Sondieren

$$s(j,k) = \lceil j/2 \rceil^2 (-1)^{j+1}$$
$$S(k) = (h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \dots) \mod m$$

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

Schlüssel 12

0	1	2	3	4	5	6

Quadratisches Sondieren

$$s(j,k) = \lceil j/2 \rceil^2 (-1)^{j+1}$$
$$S(k) = (h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \dots) \mod m$$

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

0	1	2	3	4	5	6
					12	

Quadratisches Sondieren

$$s(j,k) = \lceil j/2 \rceil^2 (-1)^{j+1}$$
$$S(k) = (h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \dots) \mod m$$

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

Schlüssel 12 , 55 , 5

0	1	2	3	4	5	6
					12	55

Quadratisches Sondieren

$s(j,k) = \lceil j/2 \rceil^2 (-1)^{j+1}$
$S(k) = (h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \dots) \mod m$

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

0	1	2	3	4	5	6
				5	12	55

Quadratisches Sondieren

$$s(j,k) = \lceil j/2 \rceil^2 (-1)^{j+1}$$
$$S(k) = (h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \dots) \mod m$$

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

Schlüssel 12 , 55 , 5 , 15 , 2

0	1	2	3	4	5	6
	15			5	12	55

Quadratisches Sondieren

$s(j,k) = \lceil j/2 \rceil^2 (-1)^{j+1}$
$S(k) = (h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \dots) \mod m$

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
	15	2		5	12	55

Quadratisches Sondieren

$$s(j,k) = \lceil j/2 \rceil^2 (-1)^{j+1}$$
$$S(k) = (h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \dots) \mod m$$

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
19	15	2		5	12	55

Analyse Quadratisches Sondieren (ohne Herleitung)

Analyse Quadratisches Sondieren (ohne Herleitung)

1. Erfolglose Suche. Durchschnittliche Anzahl betrachteter Einträge

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

Analyse Quadratisches Sondieren (ohne Herleitung)

1. Erfolglose 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 $\alpha = 0.95$

Erfolglose Suche betrachtet im Durchschnitt 22 Tabelleneinträge

Diskussion

Beispiel $\alpha = 0.95$

Erfolglose Suche betrachtet im Durchschnitt 22 Tabelleneinträge

? Grund für die schlechte Performance?

Diskussion

Beispiel $\alpha = 0.95$

Erfolglose 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.

Double Hashing

Zwei Hashfunktionen $h(k)$ und $h'(k)$. $s(j,k) = j \cdot h'(k)$.

$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \ldots, h(k) + (m-1)h'(k)) \mod m$

Double Hashing

Zwei Hashfunktionen $h(k)$ und $h'(k)$. $s(j,k) = j \cdot h'(k)$.

$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \ldots, h(k) + (m-1)h'(k)) \mod m$

Beispiel:

$m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod 7$, $h'(k) = 1 + k \bmod 5$.

Schlüssel 12

0	1	2	3	4	5	6

Double Hashing

Zwei Hashfunktionen $h(k)$ und $h'(k)$. $s(j,k) = j \cdot h'(k)$.

$$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \ldots, h(k) + (m-1)h'(k)) \mod m$$

Beispiel:

$m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod 7$, $h'(k) = 1 + k \bmod 5$.

Schlüssel 12 , 55

0	1	2	3	4	5	6
					12	

Double Hashing

Zwei Hashfunktionen $h(k)$ und $h'(k)$. $s(j,k) = j \cdot h'(k)$.

$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \ldots, h(k) + (m-1)h'(k)) \mod m$

Beispiel:

$m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \mod 7$, $h'(k) = 1 + k \mod 5$.

Schlüssel 12 , 55 , 5

0	1	2	3	4	5	6
					12	55

Double Hashing

Zwei Hashfunktionen $h(k)$ und $h'(k)$. $s(j,k) = j \cdot h'(k)$.

$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \ldots, h(k) + (m-1)h'(k)) \mod m$

Beispiel:

$m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod 7$, $h'(k) = 1 + k \bmod 5$.

Schlüssel 12 , 55 , 5 , 15

0	1	2	3	4	5	6
5					12	55

Double Hashing

Zwei Hashfunktionen $h(k)$ und $h'(k)$. $s(j,k) = j \cdot h'(k)$.

$$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \ldots, h(k) + (m-1)h'(k)) \mod m$$

Beispiel:

$m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod 7$, $h'(k) = 1 + k \bmod 5$.

Schlüssel 12 , 55 , 5 , 15 , 2

0	1	2	3	4	5	6
5	15				12	55

Double Hashing

Zwei Hashfunktionen $h(k)$ und $h'(k)$. $s(j,k) = j \cdot h'(k)$.

$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \dots, h(k) + (m-1)h'(k)) \mod m$

Beispiel:

$m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod 7$, $h'(k) = 1 + k \bmod 5$.

Schlüssel 12 , 55 , 5 , 15 , 2 , 19

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

Double Hashing

Zwei Hashfunktionen $h(k)$ und $h'(k)$. $s(j,k) = j \cdot h'(k)$.

$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \ldots, h(k) + (m-1)h'(k)) \mod m$

Beispiel:

$m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod 7$, $h'(k) = 1 + k \bmod 5$.

Schlüssel 12 , 55 , 5 , 15 , 2 , 19

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

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:

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

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

Analyse Double Hashing

Sind h und h' unabhängig, dann:

Analyse Double Hashing

Sind h und h' unabhängig, dann:

1. Erfolglose Suche. Durchschnittliche Anzahl betrachteter Einträge

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

Analyse Double Hashing

Sind h und h' unabhängig, dann:

1. Erfolglose Suche. Durchschnittliche Anzahl betrachteter Einträge

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

2. Erfolgreiche Suche. Durchschnittliche Anzahl betrachteter Einträge

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

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.

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 möglich[19]: randomisieren!

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

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

[19] Ähnlich 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 Schlüsseln 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:

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

Universelles Hashing ist relevant!

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

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

Dann gilt

Theorem

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