Datenstrukturen und Algorithmen

Vorlesung 12: Hashing

Prof. Dr. Erika Ábrahám

Theorie Hybrider Systeme
Informatik 2

http://ths.rwth-aachen.de/teaching/ss-14/
datenstrukturen-und-algorithmen/

Diese Präsentation verwendet in Teilen Folien von Joost-Pieter Katoen.

02. Juni 2014

RWTH AACHEN UNIVERSITY

Übersicht

Einführung

Dictionary (Wörterbuch)

Das Dictionary (auch: Map, assoziatives Array) speichert Informationen, die jederzeit anhand ihres Schlüssels abgerufen werden können. Weiterhin:

- Die Daten sind dynamisch gespeichert.
- `Element dictSearch(Dict d, int k)` gibt die in `d` zum Schlüssel `k` gespeicherten Informationen zurück.
- `void dictInsert(Dict d, Element e)` speichert Element `e` unter seinem Schlüssel `e.key` in `d`.
- `void dictDelete(Dict d, Element e)` löscht das Element `e` aus `d`, wobei `e` in `d` enthalten sein muss.

Beispiel

Symboltabelle eines Compilers, wobei die Schlüssel Strings (etwa Bezeichner) sind.

Einführung

Problem

Welche Datenstrukturen sind geeignet, um ein Dictionary zu implementieren?

- Heap: Einfügen und Löschen sind effizient. Aber was ist mit Suche?
- Sortiertes Array/Liste: Einfügen ist im Worst-Case linear.
- Rot-Schwarz-Baum: Alle Operationen sind im Worst-Case logarithmisch.

Lösung

Unter realistischen Annahmen benötigt eine Hashtabelle im Durchschnitt $\mathcal{O}(1)$ für die Operationen Einfügen, Löschen und Suchen.

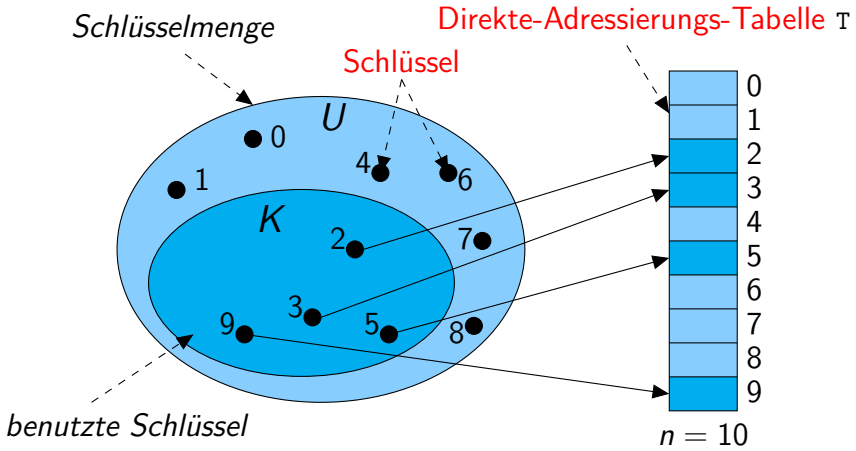
Übersicht

Direkte Adressierung

Direkte Adressierung

- Alloziere ein Array (die Direkte-Adressierungs-Tabelle), so dass es für jeden möglichen Schlüssel eine(1) Position gibt.
- Jedes Array-Element enthält einen Pointer auf die gespeicherte Information.
  - Der Einfachheit halber vernachlässigen wir in der Vorlesung die zu den Schlüsseln gehörenden Informationen.
- Mit Schlüsselmenge $U = \{0, 1, \ldots, N-1\}$ ergibt sich:
  - Eine Direkte-Adressierungs-Tabelle `T[0..N-1]`, wobei `T[k]` zu Schlüssel `k` gehört.
  - `datSearch(T, int k): return T[k];`
  - `datInsert(T, Element e): T[e.key] = e;`
  - `datDelete(T, Element e): T[e.key] = null;`
- Die Laufzeit jeder Operation ist im Worst-Case $\Theta(1)$.

Direkte Adressierung



Beispielanwendungen

Beispielanwendung: Counting Sort verwendet direkte Adressierung um die Schlüsselhäufigkeiten zu speichern.

Eine andere Anwendung: Duplikate in Linearzeit erkennen.
Gegeben seien n ganzzahlige Schlüssel zwischen 0 und $m-1$ ($m \in \Theta(n)$) in einem Array E.

```
1  bool checkDuplicates(int[] E, int n, int m) {
2    int histogram[m] = 0; // "Direkte-Adressierungs-Tabelle"
3    for (int i = 0; i < n; i++) {
4      if (histogram[E[i]] > 0) {
5        return true; // Duplikat gefunden
6      } else {
7        histogram[E[i]]++; // Zähle Häufigkeit
8      }
9    }
10   return false; // keine Duplikate
11 }
```

Nachteile der direkten Adressierung

Hauptproblem: Übermäßiger Speicherbedarf für das Array.

- Zum Beispiel bei Strings mit 20 Zeichen (5 bit/Zeichen) als Schlüssel benötigt man $2^{5 \cdot 20} = 2^{100}$ Arrayeinträge.
- Können wir diesen riesigen Speicherbedarf vermeiden und effizient bleiben? Ja! – mit Hashing.

Übersicht

Hashing

Praktisch wird nur ein kleiner Teil der Schlüssel verwendet, d. h. $|K| \ll |U|$.

$\Rightarrow$ Bei direkter Adressierung ist der größte Teil von T verschwendet.

Das Ziel von Hashing ist:

- Einen extrem großen Schlüsselraum auf einen vernünftig kleinen Bereich von ganzen Zahlen abzubilden.
- Dass zwei Schlüssel auf die selbe Zahl abgebildet werden, soll möglichst unwahrscheinlich sein.

Hashfunktion, Hashtabelle, Hashkollision

Eine Hashfunktion bildet Schlüssel auf Indices der Hashtabelle T ab:
$h : U \longrightarrow \{0, 1, \ldots, m-1\}$ für Tabellengröße m.

Wir sagen, dass $h(k)$ der Hashwert des Schlüssels k ist.
Das Auftreten von $h(k) = h(k')$ für $k \neq k'$ nennt man eine Kollision.

Example: Bucket Sort, Klausurregistrierung mit Matrikelnummern

Example: Klausurregistrierung mit Matrikelnummern

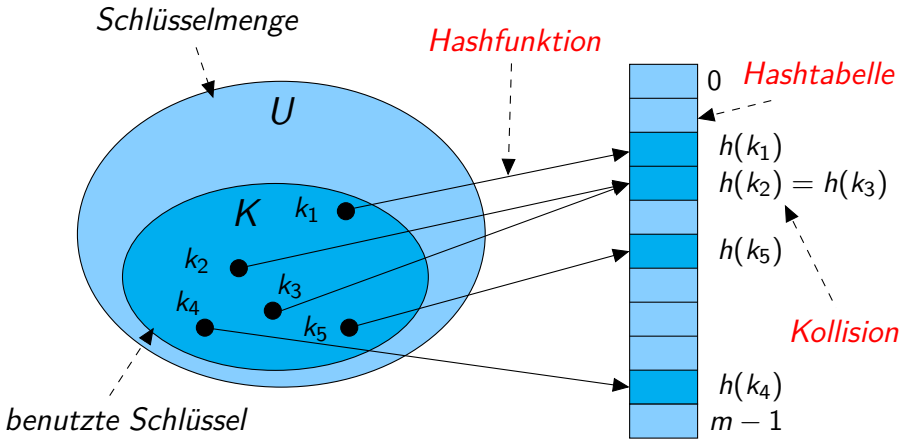
- Identifiziere Studenten über Matrikelnummern.
- 10^6 mögliche Matrikelnummern.
- In dieser Vorlesung: 600 Studenten.
- Finde eine Funktion

$$h : [0, 10^6 - 1] \longrightarrow [0, 599]\,,$$

die alle Matrikenummern aus $[0, 10^6 - 1]$ möglichst gleichverteilt auf Werte in $[0, 599]$ abbildet.

- Zum Beispiel: modulo 600.

Hashing



- Wie finden wir Hashfunktionen, die einfach auszurechnen sind und Kollisionen minimieren?
- Wie behandeln wir dennoch auftretende Kollisionen?

Kollisionen: Das Geburtstagsparadoxon

Unsere Hashfunktion mag noch so gut sein,
wir sollten auf Kollisionen vorbereitet sein!

Das liegt am

Geburtstagsparadoxon

- Die Wahrscheinlichkeit, dass dein Nachbar am selben Tag wie du Geburtstag hat ist $\frac{1}{365} \approx 0{,}027$.
- Fragt man 23 Personen, wächst die Wahrscheinlichkeit auf $\frac{23}{365} \approx 0{,}063$.
- Sind aber 23 Personen in einem Raum, dann haben zwei von ihnen den selben Geburtstag mit Wahrscheinlichkeit

$$1 - \left(\frac{365}{365} \cdot \frac{364}{365} \cdot \frac{363}{365} \cdot \ldots \cdot \frac{343}{365}\right) \approx 0{,}5$$

Kollisionen: Das Geburtstagsparadoxon

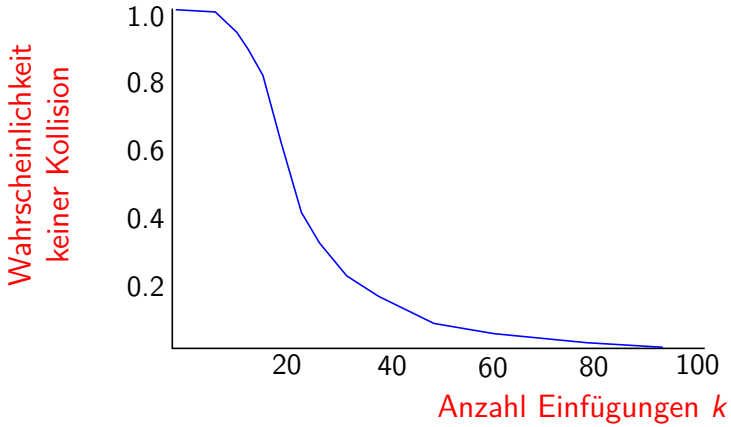
Auf Hashing angewendet bedeutet das:

- Die Wahrscheinlichkeit keiner Kollision nach n zufälligen Einfügevorgängen in einer m-elementigen Tabelle ist:

$$\frac{m}{m} \cdot \frac{m-1}{m} \cdot \ldots \cdot \frac{m-n+1}{m} = \prod_{i=0}^{n-1} \frac{m-i}{m}$$

- Dieses Produkt geht mit wachsendem n (und m) gegen 0.
- Etwa bei $m = 365$ ist die Wahrscheinlichkeit für $n \geqslant 50$ praktisch 0.

Kollisionen: Das Geburtstagsparadoxon

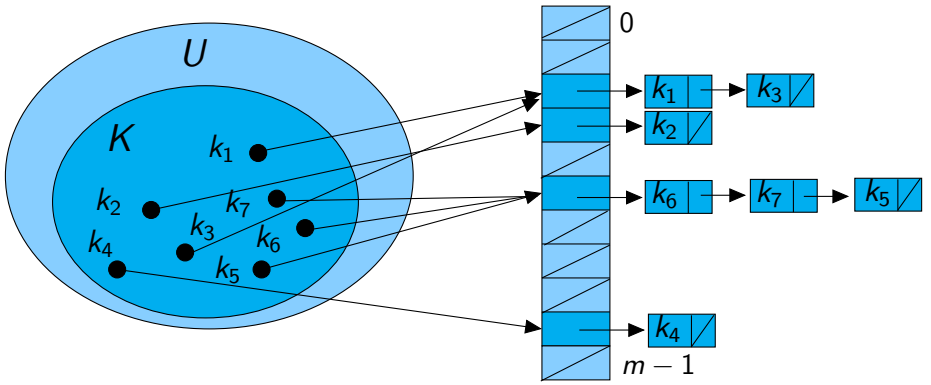


Übersicht

Kollisionsauflösung durch Verkettung

Idee

Alle Schlüssel, die zum gleichen Hash führen, werden in einer verketteten Liste gespeichert. [Luhn 1953]



Kollisionsauflösung durch Verkettung

Dictionary-Operationen bei Verkettung (informell)

- `hcSearch(int k)`: Suche nach einem Element mit Schlüssel k in der Liste `T[h(k)]`.
- `hcInsert(Element e)`: Setze Element `e` an den Anfang der Liste `T[h(e.key)]`.
- `hcDelete(Element e)`: Lösche Element `e` aus der Liste `T[h(e.key)]`.

Kollisionsauflösung durch Verkettung

Worst-Case Komplexität

Angenommen, die Berechnung von $h(k)$ ist recht effizient, etwa $\Theta(1)$.
Die Komplexität ist:

- Suche: Proportional zur Länge der Liste $T[h(k)]$.
- Einfügen: Konstant (ohne Überprüfung, ob das Element schon vorhanden ist).
- Löschen: Proportional zur Länge der Liste $T[h(k)]$.

- Im Worst-Case haben alle Schüssel den selben Hashwert.
- Suche und Löschen hat dann die selbe Worst-Case Komplexität wie Listen: $\Theta(n)$.

- Im Average-Case ist Hashing mit Verkettung aber dennoch effizient!

Average-Case-Analyse von Verkettung

Annahmen:

- Gegeben seien n aus N möglichen Schlüsseln und m Hashtabellenpositionen, $N \gg m$.
- Gleichverteiltes Hashing: Jeder Schlüssel wird mit gleicher Wahrscheinlichkeit und unabhängig von den anderen Schlüsseln auf jedes der m Slots abgebildet.
- Der Hashwert $h(k)$ kann in konstanter Zeit berechnet werden.

$\mathcal{O}$, Θ, Ω erweitert

Aus technischen Gründen erweitern wir die Definition von $\mathcal{O}$, Θ und Ω auf Funktionen mit zwei Parametern.

- Beispielsweise ist $g \in \mathcal{O}(f)$ gdw.

$$\exists c > 0, n_0, m_0 \text{ mit } \forall n \geqslant n_0, m \geqslant m_0 : 0 \leqslant g(n,m) \leqslant c \cdot f(n,m)$$

Average-Case-Analyse von Verkettung

- Der Füllgrad der Hashtabelle `T` ist $\alpha(n,m) = \frac{n}{m}$.

$\Rightarrow$ Auch die durchschnittliche Länge der Liste `T[h(k)]` ist α!

- Wieviele Elemente aus `T[h(k)]` müssen nun im Schnitt untersucht werden, um einen Schlüssel k zu finden?

$\Rightarrow$ Unterscheide erfolgreiche von erfolgloser Suche (wie in Vorlesung 1).

Average-Case-Analyse von Verkettung

Erfolglose Suche

Die erfolglose Suche benötigt $\Theta(1+\alpha)$ Zeit im Average-Case.

- Die erwartete Zeit, um Schlüssel k zu finden ist gerade die Zeit, um die Liste `T[h(k)]` zu durchsuchen.
- Die erwartete Länge dieser Liste ist α.
- Das Berechnen von $h(k)$ benötige nur eine Zeiteinheit.

$\Rightarrow$ Insgesamt erhält man $1+\alpha$ Zeiteinheiten im Durchschnitt.

Average-Case-Analyse von Verkettung

Erfolgreiche Suche

Die erfolgreiche Suche benötigt im Average-Case auch $\Theta(1+\alpha)$.

- Sei k_i der i-te eingefügte Schlüssel und $A(k_i)$ die erwartete Zeit, um k_i zu finden:

$$A(k_i) = 1 + \text{Durchschnittliche Anzahl Schlüssel, die in } \texttt{T[h(k\_i)]} \text{ erst nach } k_i \text{ eingefügt wurden}$$

- Annahme von gleichverteiltem Hashing ergibt: $A(k_i) = 1 + \sum_{j=i+1}^{n} \frac{1}{m}$
- Durchschnitt über alle n Einfügungen in die Hashtabelle: $\frac{1}{n}\sum_{i=1}^{n} A(k_i)$

Average-Case-Analyse von Verkettung

Die erwartete Anzahl an untersuchten Elementen bei einer erfolgreichen Suche ist:

$$\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) \quad \text{| Summe aufteilen}$$

$$= \frac{1}{n}\sum_{i=1}^{n}1+\frac{1}{nm}\sum_{i=1}^{n}\sum_{j=i+1}^{n}1 \quad \text{| Vereinfachen}$$

$$= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) \quad \text{| Summe } 1\ldots n-1$$

$$= 1+\frac{1}{nm}\cdot\frac{n(n-1)}{2} \quad \text{| Vereinfachen}$$

$$= 1+\frac{n-1}{2m} = 1+\frac{\alpha}{2}-\frac{\alpha}{2n} \quad \text{und damit in } \Theta(1+\alpha)$$

Komplexität der Dictionary-Operationen mit Verkettung

- Vorausgesetzt die Tabellengröße m ist (wenigstens) proportional zu N,
- dann ist der Füllgrad $\alpha(n,m) = \frac{n}{m} \in \frac{\mathcal{O}(m)}{m} = \mathcal{O}(1)$.
- Damit benötigen alle Operationen im Durchschnitt $\mathcal{O}(1)$.
- Weil das auch *Suche* mit einschließt, können wir im Average-Case mit $\mathcal{O}(n)$ sortieren.

Übersicht

Hashfunktionen

Hashfunktion

- Eine Hashfunktion bildet Schlüssel auf ganze Zahlen (d. h. Indices) ab.

- Wir interpretieren Schlüssel als natürliche Zahlen (Beispiel: Strings auf $\mathbb{N}$ abbilden).
- Was macht eine „gute“ Hashfunktion aus?
  - Die Hashfunktion $h(k)$ sollte einfach zu berechnen sein,
  - sie sollte surjektiv auf der Menge $0\ldots m-1$ sein,
  - sie sollte alle Indizes mit möglichst gleicher Häufigkeit verwenden (unabhängig von Mustern in der Schlüsselverteilung sein), und
  - ähnliche Schlüssel möglichst breit auf die Hashtabelle verteilen.
- Drei oft verwendete Techniken, eine „gute“ Hashfunktion zu erhalten:
  - Die Divisionsmethode,
  - die Multiplikationsmethode, und
  - universelles Hashing.

Divisionsmethode

Divisionsmethode

Hashfunktion: $h(k) = k \bmod m$

- Bei dieser Methode muss der Wert von m sorgfältig gewählt werden.
  - Für $m = 2^p$ ist $h(k)$ einfach die letzten p Bits.
  - Besser ist es, $h(k)$ abhängig von den Bits der Schlüsseln zu machen.
- Gute Wahl ist m prim und nicht zu nah an einer Zweierpotenz.

Beispiel

- Strings mit 2000 Zeichen als Schlüssel.
- Wir erlauben durchschnittlich 3 Schlüsseln pro Tabellenplatz.

$\Rightarrow$ Wähle $m \approx 2000/3 \longrightarrow 701$.

- Nahe an 2000/3
- Primzahl
- Nicht in der Nähe von 2er-Potenzen

Multiplikationsmethode

Multiplikationsmethode

Hashfunktion: $h(k) = \lfloor m \cdot (k \cdot c \bmod 1) \rfloor$ für $0 < c < 1$

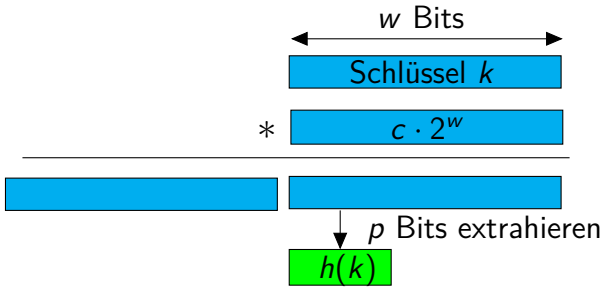
- $k \cdot c \bmod 1$ ist der Nachkommateil von $k \cdot c$, d. h. $k \cdot c - \lfloor k \cdot c \rfloor$.
- Knuth empfiehlt $c \approx (\sqrt{5} - 1)/2 \approx 0{,}62$.

$\Rightarrow$ Der Wert von m ist hier nicht kritisch.

Multiplikationsmethode

Hashfunktion: $h(k) = \lfloor m \cdot (k \cdot c \bmod 1) \rfloor$.

- Das übliche Vorgehen nimmt für w-Bit-Schlüsseln $m = 2^p$ und $c = \frac{s}{2^w}$, wobei $0 < s < 2^w$. Dann ist die Implementierung einfach:
  - Berechne zunächst $k \cdot s$ $(= k \cdot c \cdot 2^w)$.
  - Teile durch 2^w, verwende nur die Nachkommastellen.
  - Multipliziere mit 2^p und verwende nur den ganzzahligen Anteil.



Universelles Hashing

Das größte Problem beim Hashing ist,

- dass es immer eine ungünstige Sequenz von Schlüsseln gibt, die auf den selben Slot abgebildet werden.

Idee

Wähle zufällig eine Hashfunktion aus einer gegebenen kleinen Menge H, unabhängig von den verwendeten Schlüsseln.

Eine Menge Hashfunktionen ist universell, wenn

- der Anteil der Funktionen aus H, so dass k und k' kollidieren ist $\frac{|H|}{m}$.
- d. h., die W'lichkeit einer Kollision von k und k' ist $\frac{1}{|H|} \cdot \frac{|H|}{m} = \frac{1}{m}$.

Für universelles Hashing ist die erwartete Länge der Liste `T[k]`

1. Gleich α, wenn k nicht in `T` enthalten ist.
2. Gleich $1+\alpha$, wenn k in `T` enthalten ist.

Universelles Hashing

Beispiel

Definiere die Elemente der Klasse H von Hashfunktionen durch:

$$h_{a,b}(k) = ((a \cdot k + b) \bmod p) \bmod m$$

- p sei Primzahl mit $p > m$ und $p >$ größter Schlüssel.
- Die ganzen Zahlen a ($1 \leqslant a < p$) und b ($0 \leqslant b < p$) werden erst bei der Ausführung gewählt.

Die Klasse der obigen Hashfunktionen $h_{a,b}$ ist universell.

Übersicht

Offene Adressierung

- Alle Elemente werden direkt in der Hashtabelle gespeichert (im Gegensatz zur Verkettung).

$\Rightarrow$ Höchstens m Schlüssel können gespeichert werden, d. h. $\alpha(n, m) = \frac{n}{m} \leqslant 1$.

- Man spart aber den Platz für die Pointer.

Einfügen von Schlüssel k

- Sondiere (überprüfe) die Positionen der Hashtabelle in einer bestimmten Reihenfolge, bis ein leerer Slot gefunden wurde.
- Die Reihenfolge der Positionen sind vom einzufügenden Schlüssel k abgeleitet.
- Die Hashfunktion hängt also vom Schlüssel k und der Nummer der Sondierung ab:

$$h : U \times \{0, 1, \ldots m-1\} \longrightarrow \{0, 1, \ldots m-1\}$$

Einfügen bei offener Adressierung

```
1 void hashInsert(int T[], int key) {
2   for (int i = 0; i < T.length; i++) { // Teste ganze Tabelle
3     int pos = h(key, i); // Berechne i-te Sondierung
4     if (!T[pos]) { // freier Platz
5       T[pos] = key;
6       return; // fertig
7     }
8   }
9   throw "Überlauf der Hashtabelle";
10 }
```

Suche bei offener Adressierung

```
1 int hashSearch(int T[], int key) {
2   for (int i = 0; i < T.length; i++) {
3     int pos = h(key, i); // Berechne i-te Sondierung
4     if (T[pos] == key) { // Schlüssel k gefunden
5       return T[pos];
6     } else if (!T[pos]) { // freier Platz, nicht gefunden
7       break;
8     }
9   }
10  return -1; // "nicht gefunden"
11 }
```


Löschen bei offener Adressierung

Problem

Löschen des Schlüssels k aus Slot `i` durch `T[i] = null` ist ungeeignet:

- *Wenn beim Einfügen von k der Slot `i` besetzt war, können wir k nicht mehr abrufen.*

Lösung

Markiere `T[i]` mit dem speziellen Wert `DELETED` (oder: „veraltet“).

- `hashInsert` *muss angepasst werden und solche Slots als leer betrachten.*
- `hashSearch` *bleibt unverändert, solche Slots werden einfach übergangen.*

- Die Suchzeiten sind nun nicht mehr allein vom Füllgrad α abhängig.

$\Rightarrow$ Wenn Schlüssel gelöscht werden sollen wird häufiger Verkettung verwendet.

Wie wählt man die nächste Sondierung?

Wir benötigen eine Sondierungssequenz für einen gegebenen Schlüssel k:

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$$

- Wenn es sich dabei um eine Permutation von $\langle 0, \ldots m-1\rangle$ handelt ist garantiert, dass jeder Slot letztlich geprüft wird.
- Gleichverteiltes Hashing wäre ideal, d. h. jede der $m!$ Permutationen ist als Sondierungssequenz gleich wahrscheinlich.
- In der Praxis ist das aber zu aufwändig und wird approximiert.

Sondierungsverfahren

- Wir behandeln Lineares Sondieren, Quadratisches Sondieren und Doppeltes Hashing.
- Die Qualität ist durch die Anzahl der verschiedenen Sondierungssequenzen, die jeweils erzeugt werden, bestimmt.

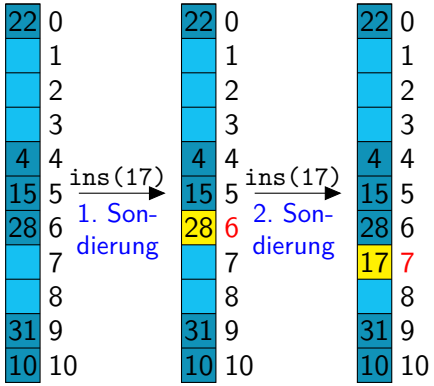
Lineares Sondieren

Hashfunktion beim linearen Sondieren

$h(k,i) = (h'(k) + i)$ **mod** m (für $i < m$).

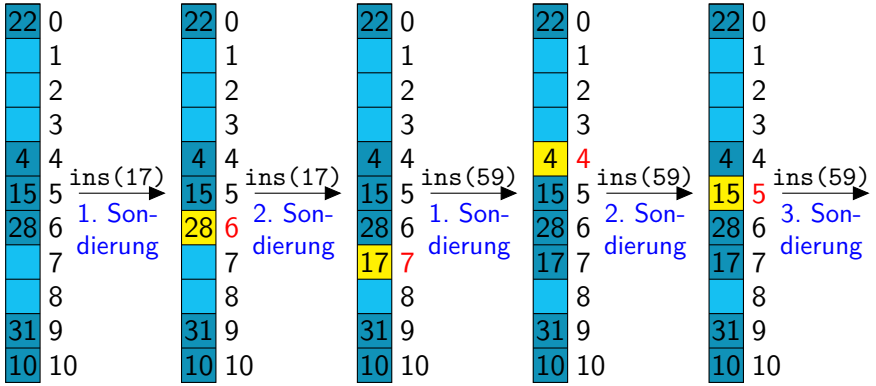
- k ist der Schlüssel
- i ist der Index im Sondierungssequenz
- h' ist eine übliche Hashfunktion.

Lineares Sondieren: Beispiel



$h'(k) = k \bmod 11$ $\quad$ $h(k,i) = (h'(k) + i) \bmod 11$

Lineares Sondieren: Beispiel



$h'(k) = k \bmod 11$ $\quad$ $h(k,i) = (h'(k) + i) \bmod 11$

Lineares Sondieren

Hashfunktion beim linearen Sondieren

$h(k,i) = (h'(k) + i) \bmod m$ (für $i < m$).

- h' ist eine übliche Hashfunktion.

- Die Verschiebung der nachfolgende Sondierungen ist linear von i abhängig.
- Die erste Sondierung bestimmt bereits die gesamte Sequenz.

$\Rightarrow$ m verschiedene Sequenzen können erzeugt werden.

- Clustering, also lange Folgen von belegten Slots, führt zu Problemen:
  - $h'(k)$ bleibt konstant, aber der Offset wird jedes Mal um eins größer.
  - Ein leerer Slot, dem i volle Slots vorausgehen, wird als nächstes mit Wahrscheinlichkeit $\frac{i+1}{m}$ gefüllt.

  $\Rightarrow$ Lange Folgen tendieren dazu länger zu werden.

Quadratisches Sondieren

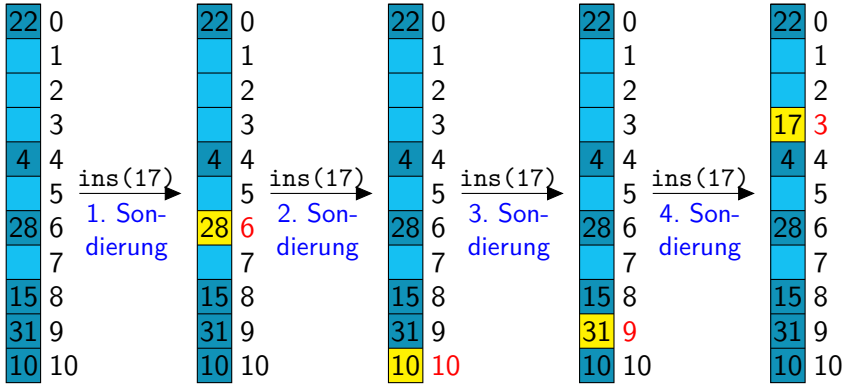
Hashfunktion beim quadratischen Sondieren

$h(k,i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$ (für $i < m$).

- k ist der Schlüssel
- i ist der Index im Sondierungssequenz
- h' ist eine übliche Hashfunktion, und
- $c_1 \in \mathbb{N}$, $c_2 \in \mathbb{N} \setminus \{0\}$ geeignete Konstanten.

Eine geeignete Wahl für die Konstanten c_1 und c_2 in Abhängigkeit von m: $m = 2^n$, $c_1 = \frac{1}{2}$, $c_2 = \frac{1}{2}$

Quadratisches Sondieren: Beispiel



$h'(k) = k \bmod 11$

$h(k,i) = (h'(k) + i + 3i^2) \bmod 11$

Quadratisches Sondieren

Hashfunktion beim quadratischen Sondieren

$h(k,i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$ (für $i < m$).

- h' ist eine übliche Hashfunktion, c_1, $c_2 \neq 0$ Konstanten.

- Die Verschiebung der nachfolgende Sondierungen ist quadratisch von i abhängig.
- Die erste Sondierung bestimmt bereits die gesamte Sequenz.

$\Rightarrow$ Auch hier können m verschiedene Sequenzen erzeugt werden (wenn c_1, c_2 geeignet gewählt wurden).

- Das Clustering von linearem Sondieren wird vermieden.
- Jedoch tritt *sekundäres* Clustering immer noch auf:

$h(k,0) = h(k',0)$ verursacht $h(k,i) = h(k',i)$ für alle i.

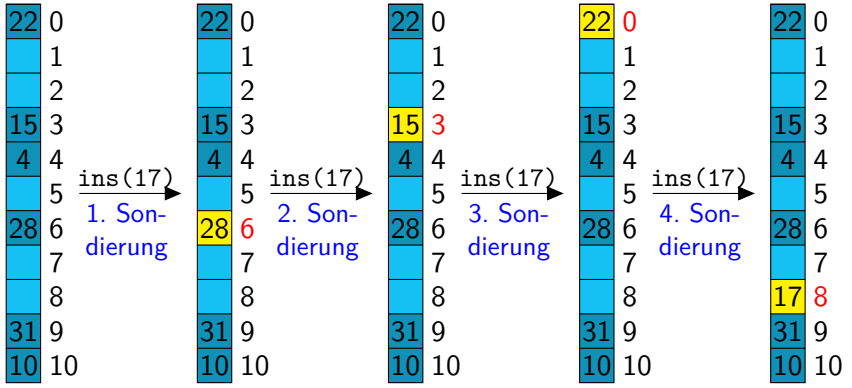
Doppeltes Hashing

Hashfunktion beim doppelten Hashing

$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m$ (für $i < m$).

- h_1, h_2 sind übliche Hashfunktionen.

Doppeltes Hashing: Beispiel



$h_1(k) = k \bmod 11$
$h_2(k) = 1 + k \bmod 10$

$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod 11$

Doppeltes Hashing

Hashfunktion beim doppelten Hashing

$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m$ (für $i < m$).

- h_1, h_2 sind übliche Hashfunktionen.

- Die erste Sondierung bestimmt nicht die gesamte Sequenz.
- Die Verschiebung der nachfolgende Sondierungen ist von $h_2(k)$ abhängig.

$\Rightarrow$ Bessere Verteilung der Schlüssel in der Hashtabelle.

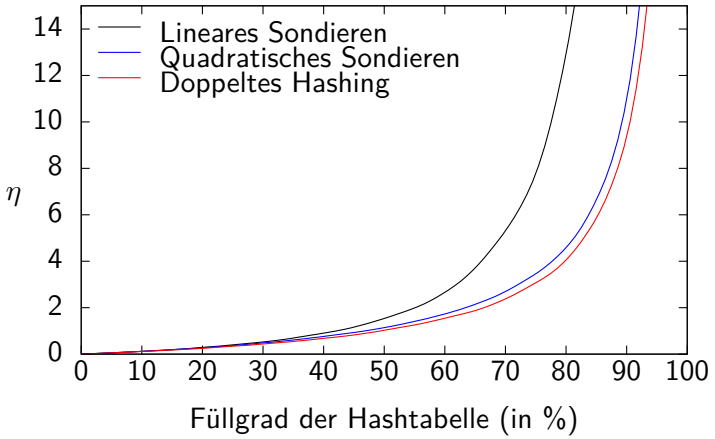
$\Rightarrow$ Approximiert das gleichverteilte Hashing.

- Sind $h_2(k)$ und m teilerfremd, wird die gesamte Hashtabelle abgesucht.
  - Wähle z. B. $m = 2^k$ und h_2 so, dass sie nur ungerade Zahlen erzeugt.
- Jedes mögliche Paar $h_1(k)$ und $h_2(k)$ erzeugt eine andere Sequenz.

$\Rightarrow$ Daher können m^2 verschiedene Permutationen erzeugt werden.

Praktische Effizienz vom doppelten Hashing

- Hashtabelle mit 538 051 Einträgen
- *Mittlere* Anzahl Kollisionen η pro Einfügen in die Hashtabelle:



Effizienz der offenen Adressierung

Unter der Annahme von gleichverteiltem Hashing gilt:

Erfolglose Suche

Die erfolglose Suche benötigt $O\left(\frac{1}{1-\alpha}\right)$ Zeit im Average-Case.

- Bei 50% Füllung sind durchschnittlich 2 Sondierungen nötig.
- Bei 90% Füllung sind durchschnittlich 10 Sondierungen nötig.

Erfolgreiche Suche

Die erfolgreiche Suche benötigt $O\left(\frac{1}{\alpha} \cdot \ln \frac{1}{1-\alpha}\right)$ im Average-Case.

- Bei 50% Füllung sind durchschnittlich 1,39 Sondierungen nötig.
- Bei 90% Füllung sind durchschnittlich 2,56 Sondierungen nötig.

Bei der Verkettung hatten wir $\Theta(1+\alpha)$ in beiden Fällen erhalten.

Analyse der erfolglosen Suche

Annahmen

- Betrachte eine zufällig erzeugte Sondierungssequenz für Schlüssel k.
- Annahme: jede mögliche Sondierungssequenz hat die gleiche Wahrscheinlichkeit $\frac{1}{m!}$, da es $m!$ mögliche Permutationen von den Positionen $0, \ldots, m-1$ gibt.
- Bemerkung: dies ist nicht unrealistisch, da im Idealfall die Sondierungssequenz für k möglichst unabhängig ist von der Sondierungssequenz für k', $k \neq k'$.
- Wir nehmen (wie vorher) an, dass die Berechung von Hashwerten in $O(1)$ liegt.

Analyse der erfolglosen Suche

Erfolglose Suche

Eine Suche für k ist erfolglos wenn für i alle Slots $h(k,0),\ldots,h(k,i-1)$ belegt sind, jedoch unterschiedlich von k sind, und $h(k,i)$ ist unbelegt.

Sei X die Anzahl der belegten Positionen bis eine freie Position gefunden wird:

$$X \;=\; \min\{\, i \in \mathbb{N} \mid h(k,i) \text{ ist unbelegt} \,\}.$$

Sei $E[X]$ der Erwartungswert von X.
Dann: die Average-Case Komplexität einer erfolglosen Suche ist $1 + E[X]$.

Lemma

$1 + E[X] \in O\left(\frac{1}{1-\alpha}\right)$.

Beweis: im Buch.

Analyse der erfolgreichen Suche

- Sei Schlüssel k_i der i-te eingefügte Schlüssel in der Hashtabelle.
- Sei X_i die Anzahl der Sondierungen beim Einfügen vom Schlüssel k_i.
- Die Suche für k_i braucht im Mittel $E[X_i]$ Zeiteinheiten.
- Die Average-Case Zeitkomplexität für eine erfolgreiche Suche ist:

$$\frac{1}{n}\sum_{i=0}^{n-1} E[X_{i+1}].$$

Lemma

$\frac{1}{n}\sum_{i=0}^{n-1} E[X_{i+1}] \;\in\; O\left(\frac{1}{\alpha}\cdot \ln\frac{1}{1-\alpha}\right)$.

Beweis: im Buch.