

# Datenstrukturen und Algorithmen

## Vorlesung 13: Hashing II

Joost-Pieter Katoen

Lehrstuhl für Informatik 2  
Software Modeling and Verification Group

<http://www-i2.informatik.rwth-aachen.de/i2/dsal12/>

05. Juni 2012



## Übersicht

### 1 Hashing und Verkettung

### 2 Offene Adressierung

- Lineares Sondieren
- Quadratisches Sondieren
- Doppeltes Hashing
- Effizienz der offenen Adressierung

## Übersicht

### 1 Hashing und Verkettung

### 2 Offene Adressierung

- Lineares Sondieren
- Quadratisches Sondieren
- Doppeltes Hashing
- Effizienz der offenen Adressierung

## Hashing

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

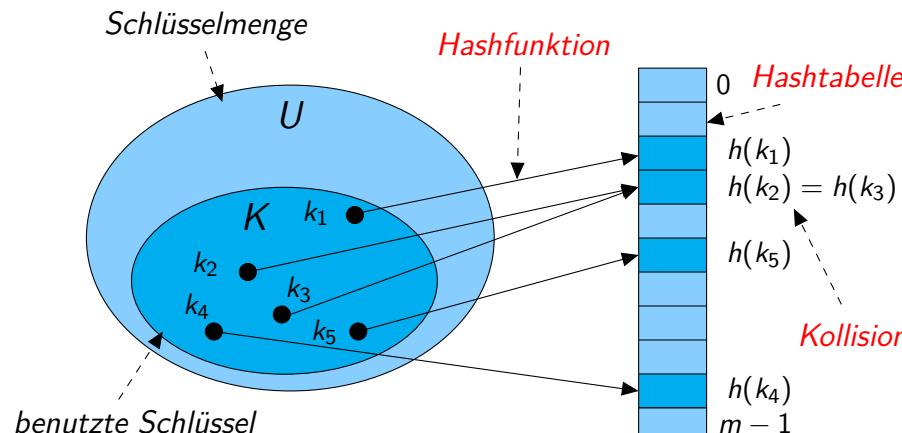
Eine **Hashfunktion** bildet einen Schlüssel auf einen Index der **Hashtabelle** ab:

$$h : U \longrightarrow \{0, 1, \dots, m-1\} \text{ für Tabellengröße } m \text{ und } |U| = n.$$

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**.

## Hashing



- ▶ Wie behandeln wir dennoch auftretende Kollisionen?
- ▶ Zwei Varianten: Verkettung und Offene Adressierung

## Average-Case-Analyse von Verkettung

### Erfolgreiche Suche

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

### Erfolglose Suche

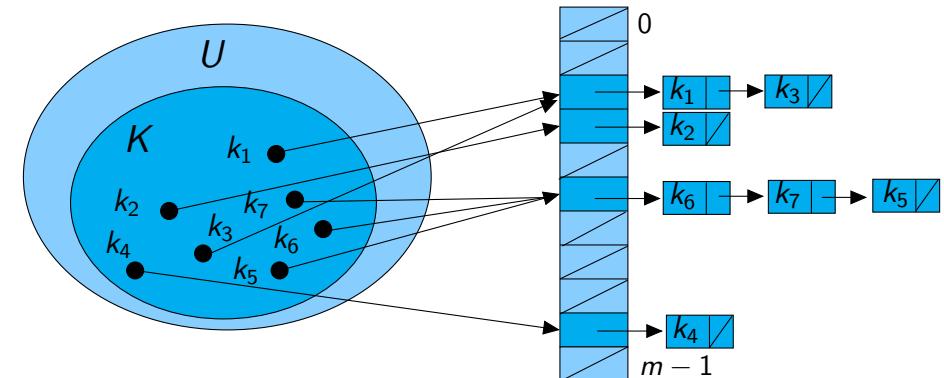
Die erfolgreiche Suche benötigt im Average-Case auch  $\Theta(1 + \alpha)$ .

## Kollisionsauflösung durch Verkettung (I)

### Idee

Alle Schlüssel, die zum gleichen Hash führen, werden in einer **verketteten Liste** gespeichert.

[Luhn 1953]



## Übersicht

### 1 Hashing und Verkettung

### 2 Offene Adressierung

- Lineares Sondieren
- Quadratisches Sondieren
- Doppeltes Hashing
- Effizienz der offenen Adressierung

## Kollisionsauflösung durch offene Adressierung

- Alle Elemente werden direkt in der Hashtabelle gespeichert (im Gegensatz zur Verkettung).
- ⇒ Höchstens  $n$  Schlüssel können gespeichert werden, d. h.  $\alpha(n, m) = \frac{n}{m} \leq 1$ . [Amdahl 1954]
- Man spart aber den Platz für die Pointer.

### Einfügen von Schlüssel $k$

- Sondiere (to probe) die Positionen der Hashtabelle, bis ein leerer Slot gefunden wurde.
- Die zu überprüfenden Positionen sind vom einzufügenden Schlüssel  $k$  abgeleitet.
- Die Hashfunktion hängt also vom Schlüssel  $k$  und der **Nummer der Sondierung** ab:

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

## 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 }
```

## 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 }
```

## Löschen bei offener Adressierung

### Problem

Löschen des Schlüssels  $k$  aus Slot  $i$  durch  $T[i] = \text{null}$  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  $\alpha$  abhängig.
- ⇒ Wenn Schlüssel gelöscht werden sollen wird häufiger **Verkettung** verwendet.

## Wie wählt man die nächste Sondierung?

Wir benötigen eine **Sondierungssequenz** für einen gegebenen Schlüssel  $k$ :

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

- Wenn es sich dabei um eine Permutation von  $\langle 0, \dots, m-1 \rangle$  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.

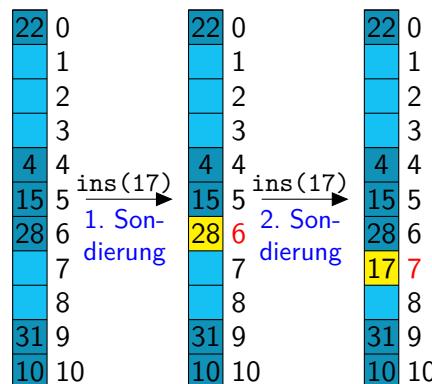
## Lineares Sondieren

### Hashfunktion beim linearen Sondieren

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

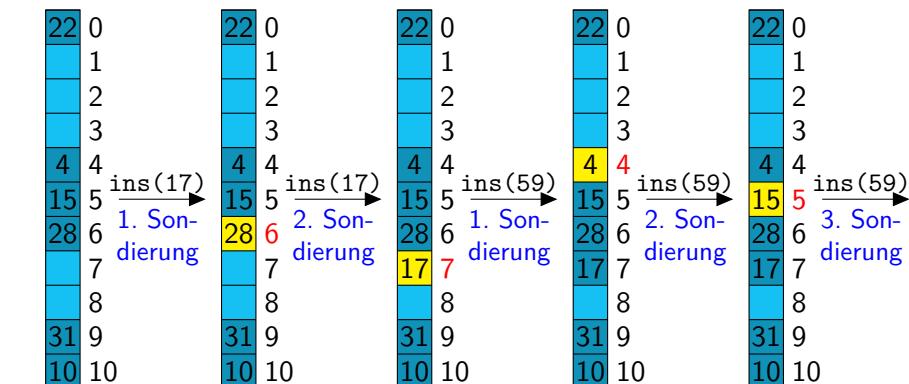
- $k$  ist der Schlüssel
- $i$  ist der Index im Sondierungssequenz
- $h'$  ist eine übliche Hashfunktion.

## Lineares Sondieren: Beispiel



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

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



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

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

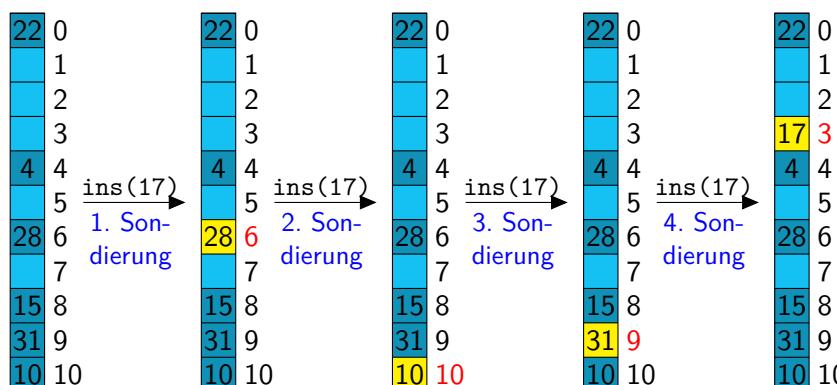
## Lineares Sondieren

### Hashfunktion beim linearen Sondieren

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

- $h'$  ist eine übliche Hashfunktion.
- Die Verschiebung der nachfolgende Sondierungen ist linear von  $i$  abhängig.
- Die erste Sondierung bestimmt bereits die gesamte Sequenz.
- ⇒  $m$  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.
- ⇒ Lange Folgen tendieren dazu länger zu werden.

## Quadratisches Sondieren: Beispiel



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

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

## Quadratisches Sondieren

### Hashfunktion beim quadratischen Sondieren

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

- $k$  ist der Schlüssel
- $i$  ist der Index im Sondierungssequenz
- $h'$  ist eine übliche Hashfunktion, und
- $c_1, c_2 \neq 0$  Konstanten.

## Quadratisches Sondieren

### Hashfunktion beim quadratischen Sondieren

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

- $h'$  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.
- ⇒ 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:

$$h(k, 0) = h(k', 0) \text{ verursacht } h(k, i) = h(k', i) \text{ für alle } i.$$

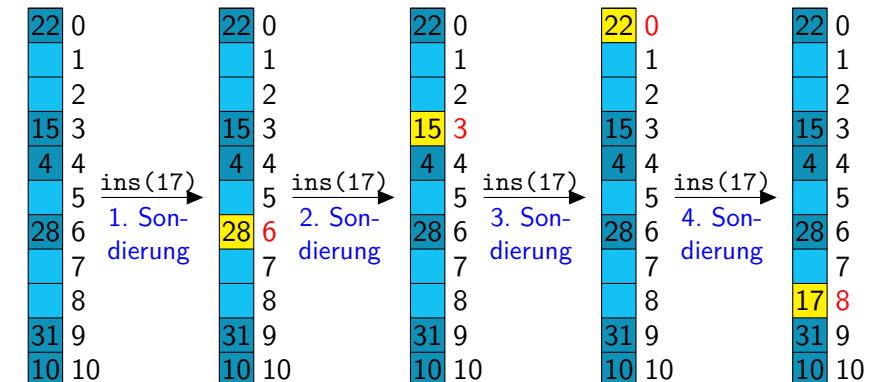
## Doppeltes Hashing

### Hashfunktion beim doppelten Hashing

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

- $h_1, h_2$  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

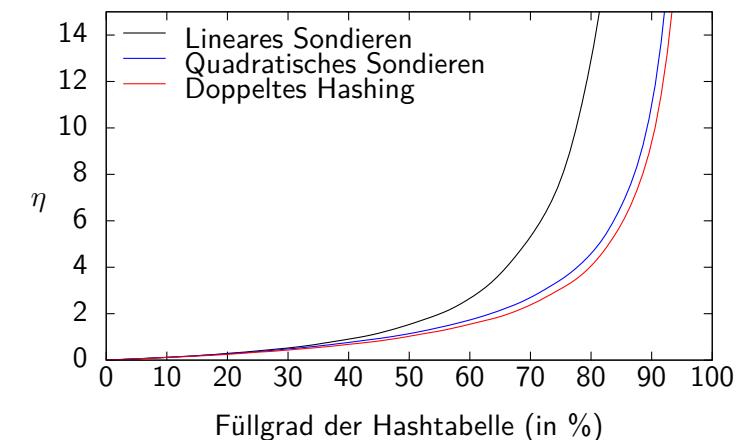
### Hashfunktion beim doppelten Hashing

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

- $h_1, h_2$  sind übliche Hashfunktionen.
- Die Verschiebung der nachfolgende Sondierungen ist von  $h_2(k)$  abhängig.
- Die erste Sondierung bestimmt **nicht** die gesamte Sequenz.
- ⇒ Bessere Verteilung der Schlüssel in der Hashtabelle.
- ⇒ Approximiert das gleichverteilte Hashing.
- Sind  $h_2$  und  $m$  relativ prim, 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.
- ⇒ Daher können  $m^2$  verschiedene Permutationen erzeugt werden.

## Praktische Effizienz von Doppeltem Hashing

- Hashtabelle mit 538 051 Einträgen (Endfüllgrad 99,95%) 99,8 % -> 358
- **Mittlere Anzahl Kollisionen  $\eta$  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 (II)

### Erfolglose Suche

Eine Suche für  $k$  ist **erfolglos** wenn für  $i$  alle Slots  $h(k, 0), \dots, h(k, i-1)$  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 ist:

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

Sei  $E[X]$  der Erwartungswert von  $X$ .

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

### Lemma

$$E[X] = \frac{n}{m-n+1}.$$

Beweis: in der Vorlesung. Damit:  $1 + E[X] = 1 + \frac{n}{m-n+1} \in O\left(\frac{1}{1-\alpha}\right)$ .

## Analyse der erfolglosen Suche (I)

### Annahmen

- Betrachte eine **zufällig** erzeugte Sondierungssequenz für Schlüssel  $k$ .
- Annahme: jede mögliche Sondierungssequenz hat eine **gleiche Wahrscheinlichkeit**, d. h.  $\frac{1}{m!}$ , da es  $m!$  mögliche Permutationen von den Positionen  $0, \dots, m-1$  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 Berechnung von Hashwerten in  $O(1)$  liegt.

## Analyse der erfolgreichen Suche (I)

- Sei Schlüssel  $k_i$  der  $i$ -te eingefügte Schlüssel in der Hashtabelle.
- Betrachte eine erfolgreiche Suche für Schlüssel  $k_{i+1}$ .
- Sei  $X_i$  die Anzahl der Sondierungen beim Einfügen vom Schlüssel  $k_i$ .
- Im Schnitt, braucht eine erfolgreiche Suche für  $k_i$ ,  $E[X_i]$  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: in der Vorlesung.