# Datenstrukturen und Algorithmen

## Vorlesung 13: Hashing II

Joost-Pieter Katoen

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

https://moves.rwth-aachen.de/teaching/ss-18/dsal/

11. Juni 2018

RWTH AACHEN UNIVERSITY

# Übersicht

# Übersicht

# Hashing

Das Ziel von Hashing ist:

- Einen extrem großen Schlüsselraum auf einen vernünftig kleinen Bereich von ganzen Zahlen abzubilden.

# 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.

# 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 T ab:

$$h : U \longrightarrow \{0, 1, \ldots, m-1\} \text{ für Tabellengröße } m \text{ und } |U| = N$$

univ. der Schlüssel

$m \ll N$

$h(i) = r \qquad 0 \leq r < m \qquad T[r]$

# 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 `T` ab:

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

Wir sagen, dass $h(k)$ der Hashwert des Schlüssels $k$ ist.

# 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.

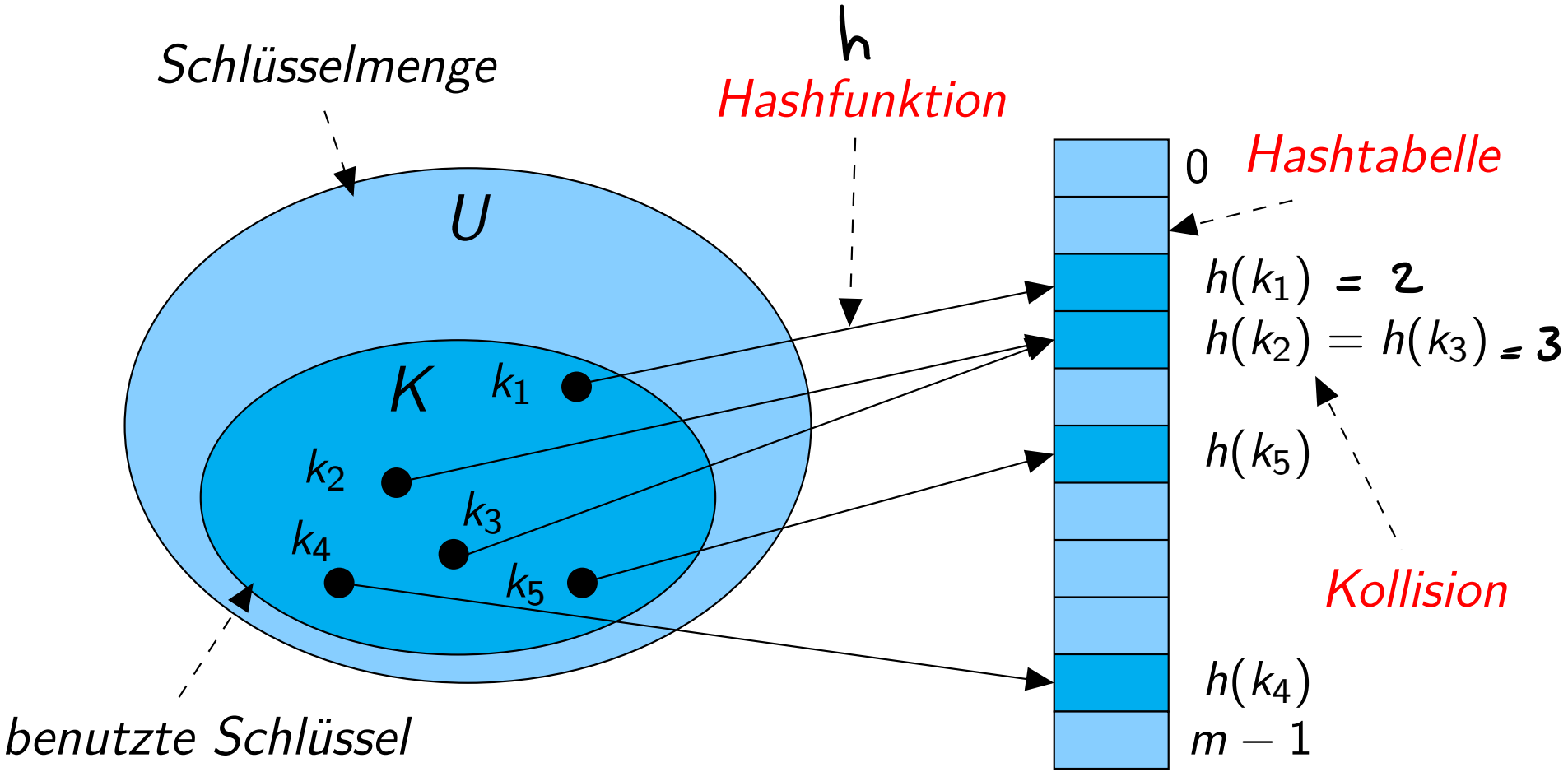
## Hashfunktion, Hashtabelle, Hashkollision

Eine Hashfunktion bildet einen Schlüssel auf einen Index der Hashtabelle `T` ab:

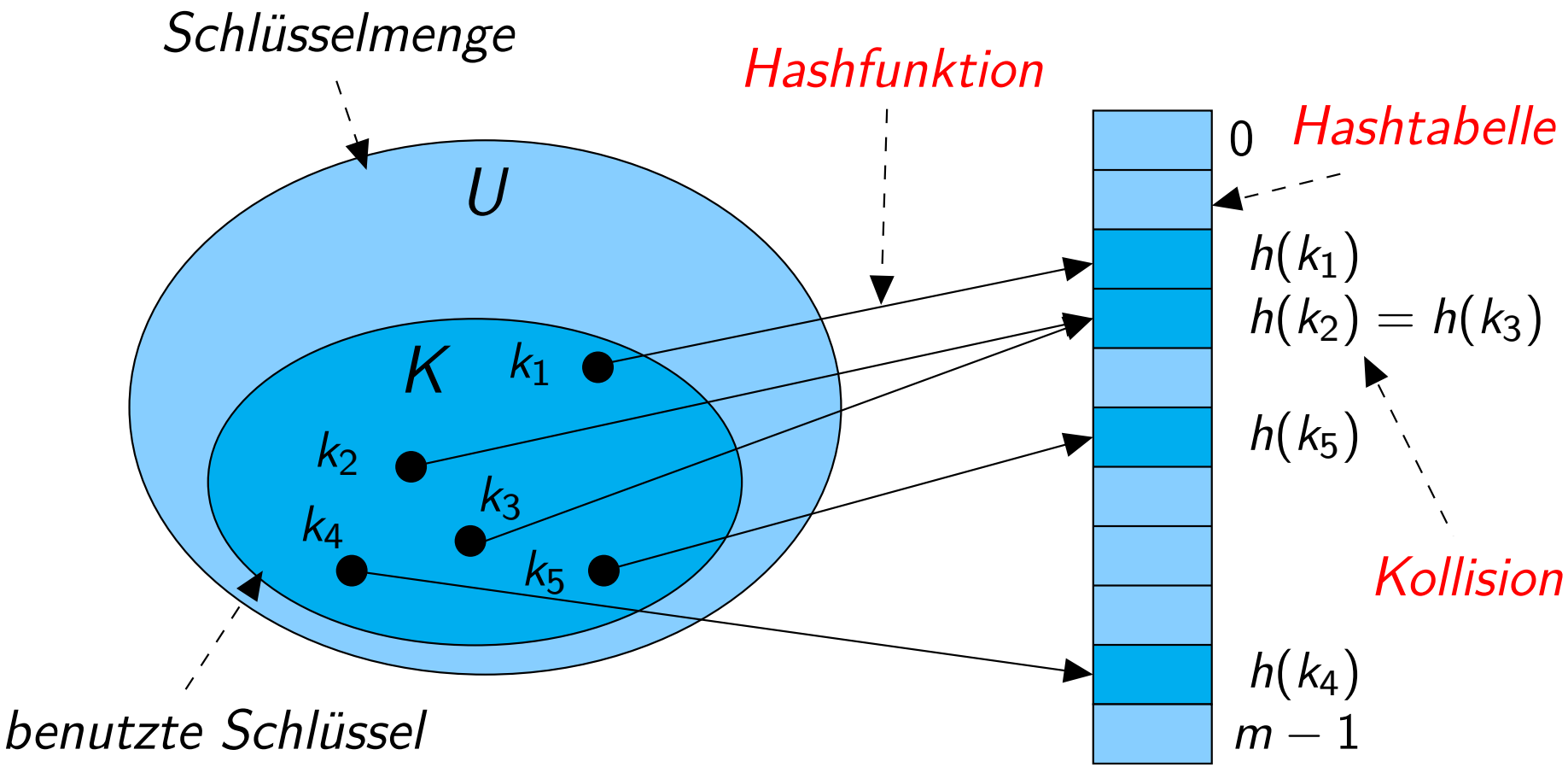
$$h : U \longrightarrow \{0, 1, \ldots, m-1\} \text{ für Tabellengröße } m \text{ und } |U| = n$$

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


Schlüsselmenge
h
Hashfunktion
U
K
$k_1$
$k_2$
$k_3$
$k_4$
$k_5$
0
Hashtabelle
$h(k_1) = 2$
$h(k_2) = h(k_3) = 3$
$h(k_5)$
Kollision
$h(k_4)$
$m-1$
benutzte Schlüssel

# Hashing




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

# Kollisionsauflösung durch Verkettung (I)

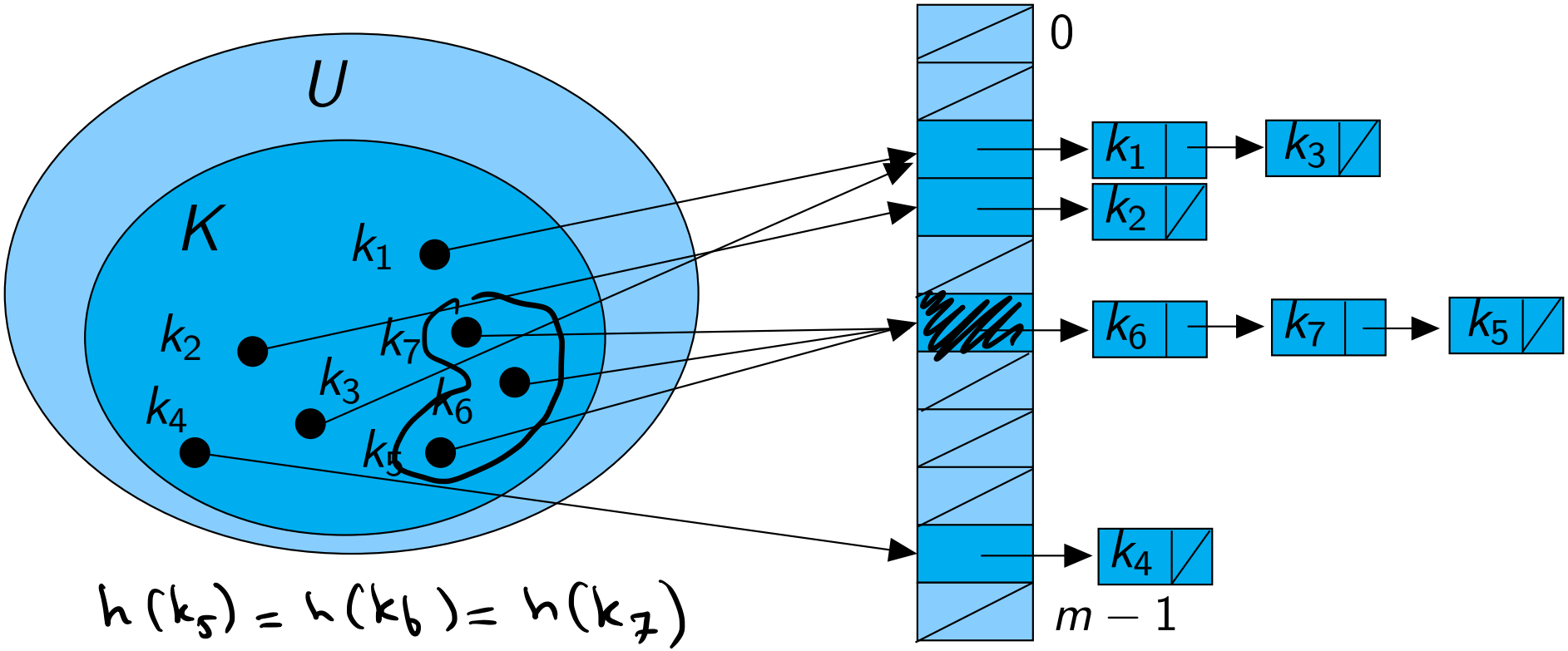
## Idee

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

# Kollisionsauflösung durch Verkettung (I)

## Idee

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



# Average-Case-Analyse von Verkettung

## Erfolgreiche Suche

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

$\alpha = \frac{n}{m}$ Füllgrad

$n$ = Anzahl belegter Slots in $T$

$|T| = m \ll N = |U|$

# Average-Case-Analyse von Verkettung

$\Theta(1)$

## Erfolgreiche Suche

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

## Erfolglose Suche

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

$\alpha = \frac{n}{m}$ $\quad n \in O(m)$

# Übersicht

# Kollisionsauflösung durch offene Adressierung

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

# Kollisionsauflösung durch offene Adressierung

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

$\Rightarrow$ Höchstens $m$ Schlüssel können gespeichert werden, d. h. $\alpha(n, m) = \frac{n}{m} \leqslant 1$. [Amdahl 1954]

# Kollisionsauflösung durch offene Adressierung

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

$\Rightarrow$ Höchstens $m$ Schlüssel können gespeichert werden, d. h. $\alpha(n, m) = \frac{n}{m} \leqslant 1$. [Amdahl 1954]

- Man spart aber den Platz für die Pointer.

# Kollisionsauflösung durch offene Adressierung

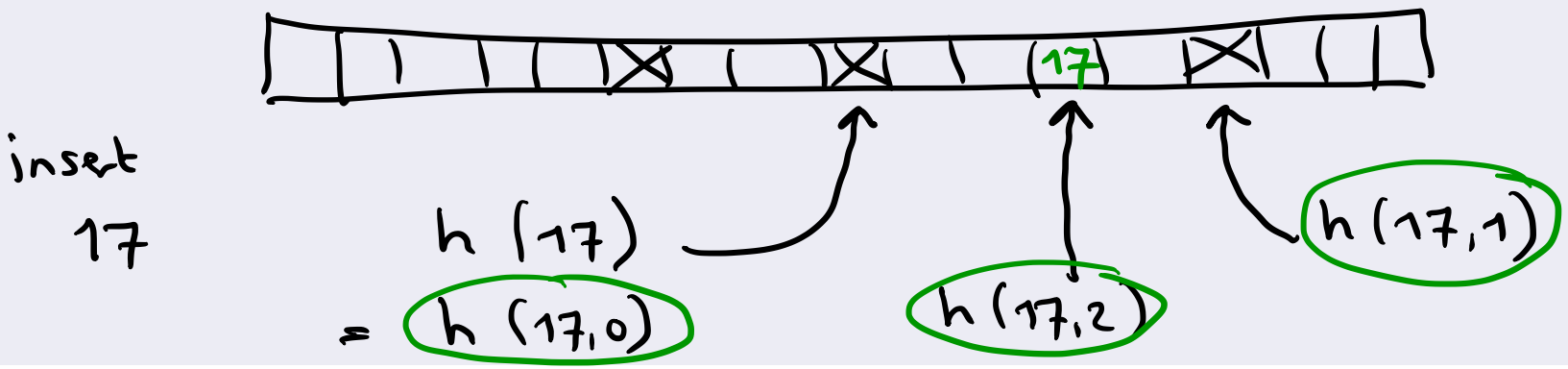
- Alle Elemente werden direkt in der Hashtabelle gespeichert (im Gegensatz zur Verkettung).

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



# Kollisionsauflösung durch offene Adressierung

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

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

17

# Kollisionsauflösung durch offene Adressierung

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

$\Rightarrow$ Höchstens $m$ Schlüssel können gespeichert werden, d. h. $\alpha(n, m) = \frac{n}{m} \leqslant 1$. [Amdahl 1954]

- Man spart aber den Platz für die Pointer.

## Einfügen von Schlüssel $k$

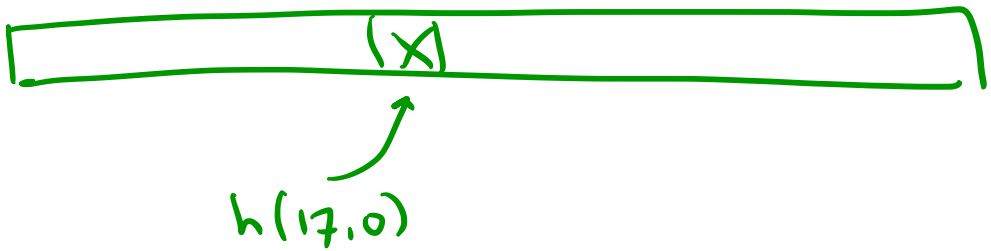
- Sondiere (to probe) die Positionen der Hashtabelle, bis ein leerer Slot gefunden wurde.
- Die zu überprüfenden Positionen sind vom einzufügenden Schlüssel $k$ abgeleitet. $h(17,0)$ $h(17,1)$ $h(17,2)$
- Die Hashfunktion hängt also vom Schlüssel $k$ und der Nummer der Sondierung ab: $h(17, m-1)$

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

# 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 }
```

h(17,0)
h(17,1)
⋮



# 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 }
```

# Löschen bei offener Adressierung

# Löschen bei offener Adressierung

## Problem

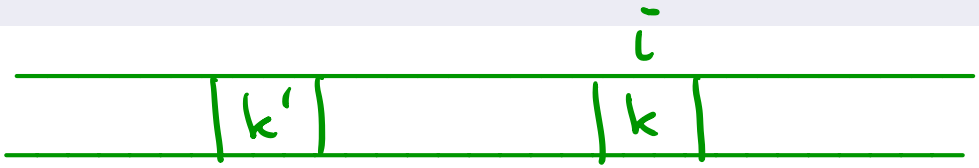
*Löschen des Schlüssels k aus Slot* `i` *durch* `T[i] = null` *ist ungeeignet:*

# 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.*




$h(k) = i$

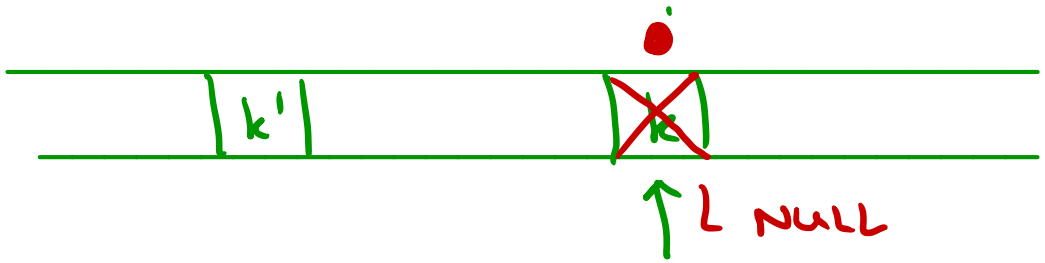
sei $k'$ nach $k$ eingefügt in $T$

und beim Einfüge von $k'$ wurde mal sondiert

so daß $h(k', j) = i$

$h(k', j+1)$

$h(k', j+2)$




suche $k'$ nachdem $k$ gelöscht wurde

# Löschen bei offener Adressierung

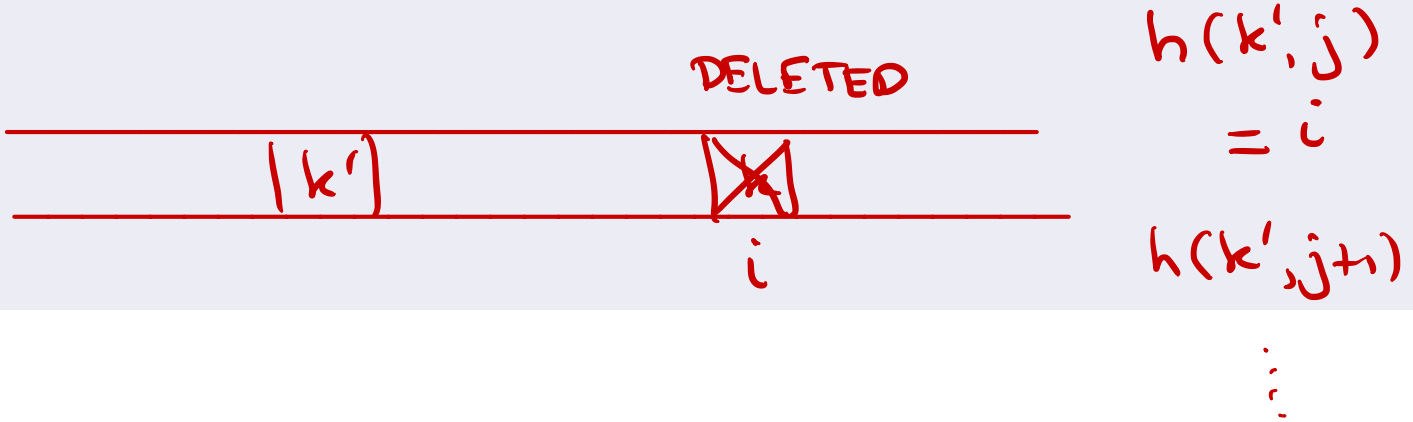
## Problem

*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“).*



# 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.*

# 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.*

# 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 $\alpha$ abhängig.

# 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 $\alpha$ abhängig.

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

# Wie wählt man die nächste Sondierung?

$h(17,0)$

$h(17,1)$

$h(17,2)$

$\vdots$

# Wie wählt man die nächste Sondierung?

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

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

m Positionen in T

# 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.

# 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.

m Positionen

m! Permutationen

$$Pr(\text{Permutation } x) = \frac{1}{m!}$$

# 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.

# 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

# 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.

# 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

$h'(k) + i$ — Linear

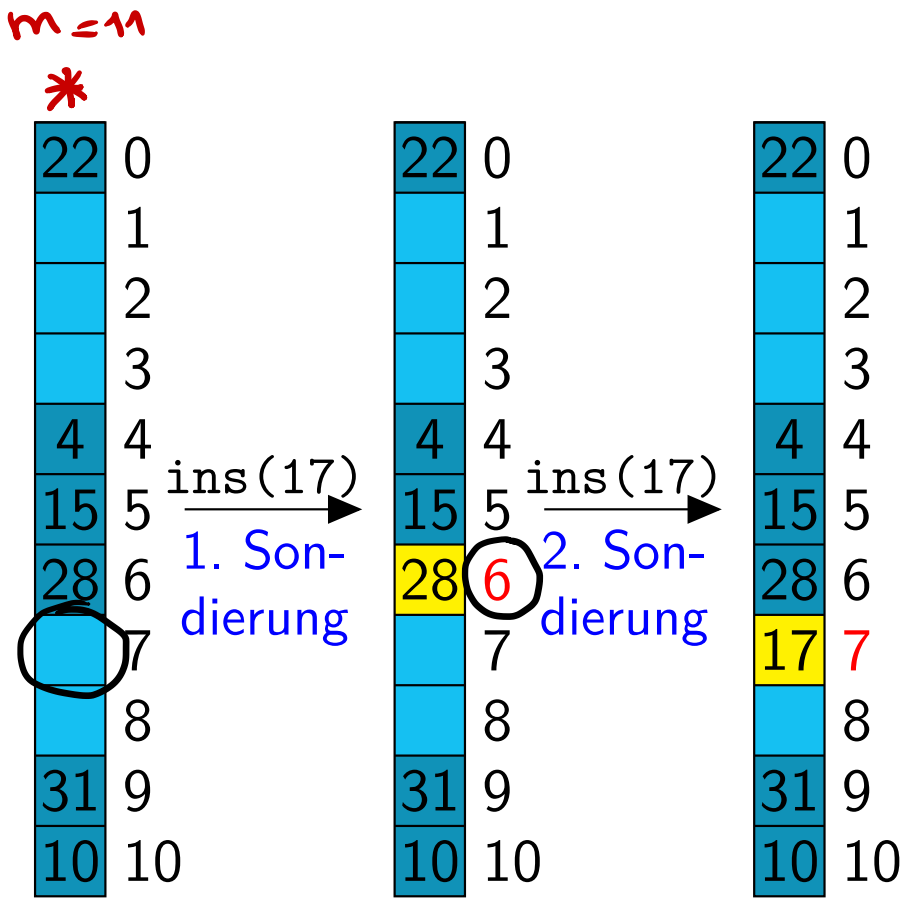
$\{0, \ldots, m-1\}$

## Hashfunktion beim linearen Sondieren

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

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

# Lineares Sondieren: Beispiel




$h(17,0) =$

$(\underbrace{h'(17)}_{6} + 0) \bmod 11$

$6 = 6$

$h(6) = 28$

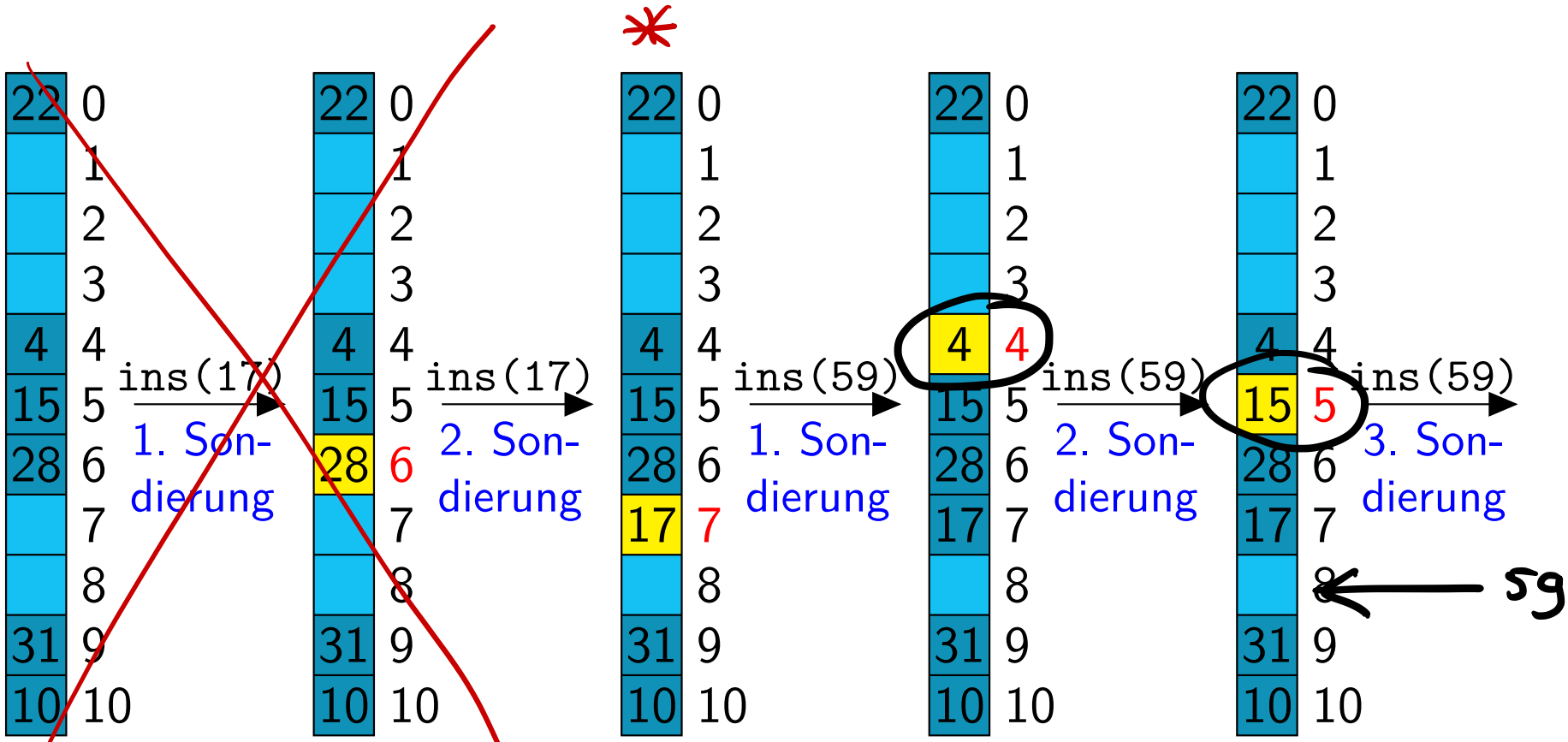
$h(17,1) =$

$(6+1) \bmod 11 = 7$

$$h'(k) = k \textbf{ mod } 11$$

$$h(k,i) = (h'(k) + i) \textbf{ mod } 11$$

# Lineares Sondieren: Beispiel




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

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

$$h(59,0) = (\underbrace{h'(59)}_{59 \bmod 11} + 0) \bmod 11$$

$$= 4$$

---

$$h(59,1) = (4+1) \bmod 11$$

$$= 5$$

---

$$h(59,2) = 6$$

$$h(59,3) = 7$$

$$h(59,4) = 8 \quad \ddot{\smile}$$

# Lineares Sondieren

## Hashfunktion beim linearen Sondieren

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

- $h'$ ist eine übliche Hashfunktion.

# Lineares Sondieren

## Hashfunktion beim linearen Sondieren

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

- $h'$ ist eine übliche Hashfunktion.

- Die Verschiebung der nachfolgende Sondierungen ist linear von $i$ abhängig.

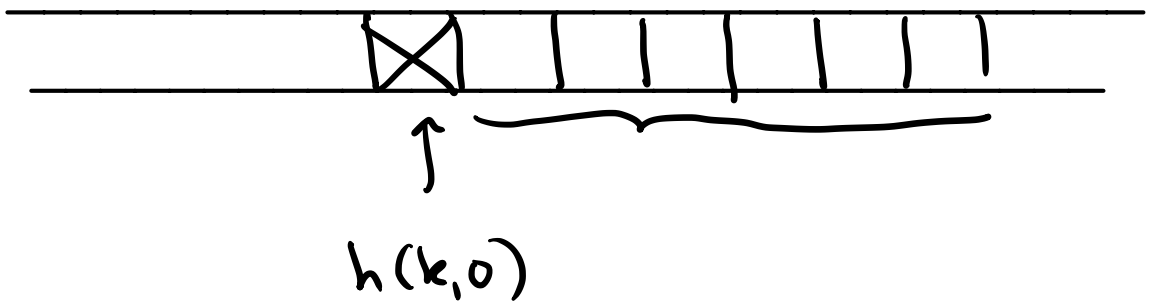
# Lineares Sondieren

## Hashfunktion beim linearen Sondieren

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



# Lineares Sondieren

## Hashfunktion beim linearen Sondieren

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

$=$

$m!$

# Lineares Sondieren

## Hashfunktion beim linearen Sondieren

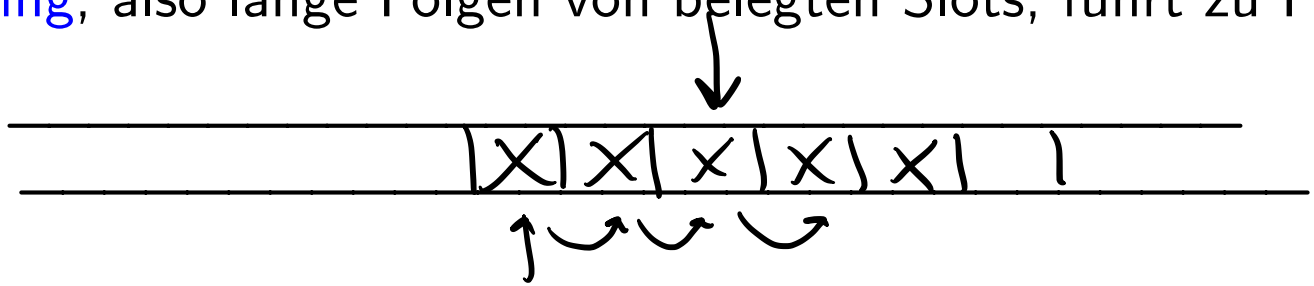
$h(k, i) = (h'(k) + i)$ **mod** $m$ (für $i < m$).

- $h'$ 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:



# Lineares Sondieren

## Hashfunktion beim linearen Sondieren

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

# Lineares Sondieren

## Hashfunktion beim linearen Sondieren

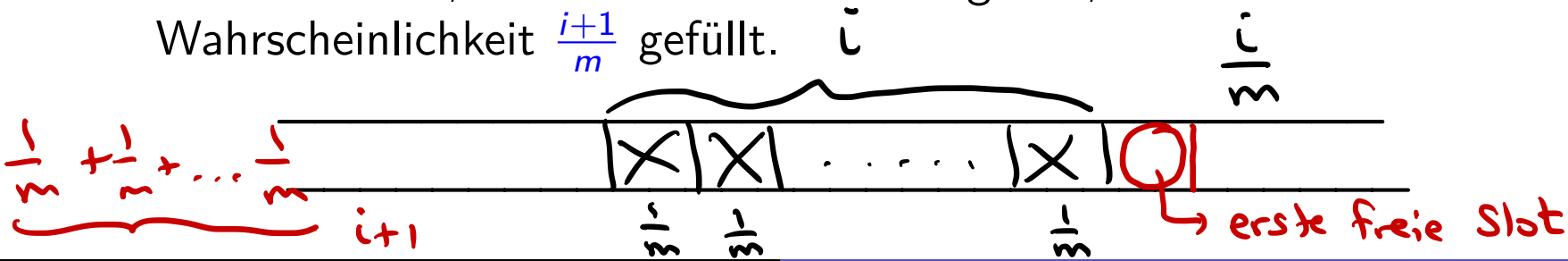
$h(k, i) = (h'(k) + i)$ **mod** $m$ (für $i < m$).

- $h'$ 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.



# Lineares Sondieren

## Hashfunktion beim linearen Sondieren

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

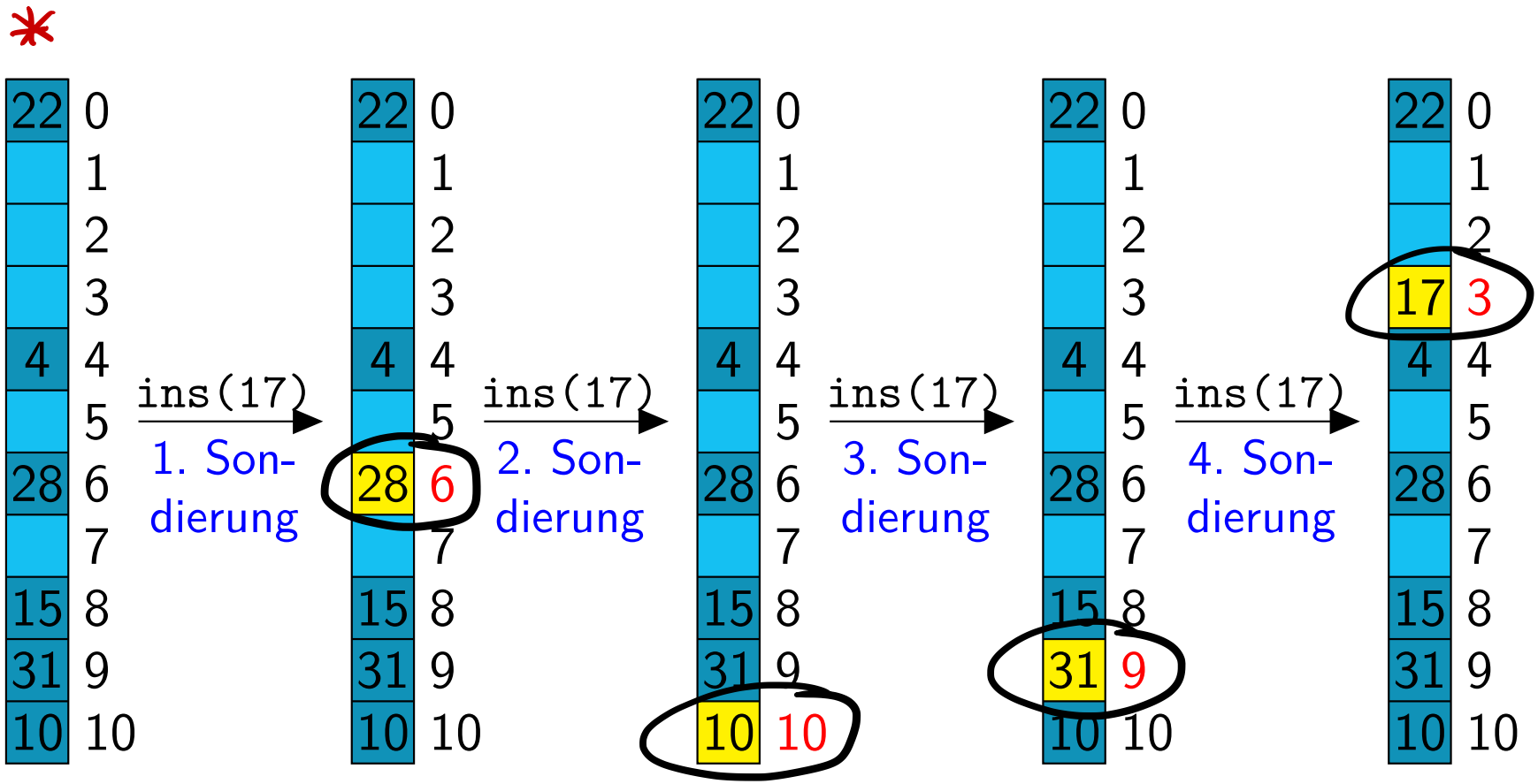
# Quadratisches Sondieren

## 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
- $c_1, c_2 \in \mathbb{N} \setminus \{0\}$ geeignete Konstanten.

# Quadratisches Sondieren: Beispiel




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

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

$c_1 = 1 \quad c_2 = 3$

~~h(17,0)~~

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

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

$h(17,0) = 6 \bmod 11 = 6$ → 10

$h(17,1) = (6 + 1 + 3) \bmod 11 = 10$

$h(17,2) = (6 + 2 + 12) \bmod 11 = 9$

$h(17,3) = (6 + 3 + 3 \cdot 3^2) \bmod 11 = 3$

# Quadratisches Sondieren

## Hashfunktion beim quadratischen Sondieren

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

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

# Quadratisches Sondieren

## Hashfunktion beim quadratischen Sondieren

$h(k, i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2)$ **mod** $m$ (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.

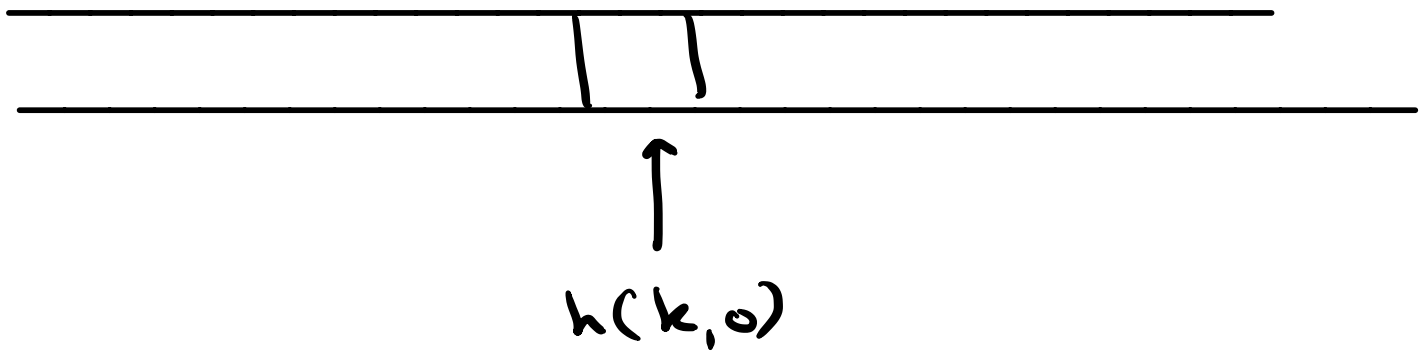
# Quadratisches Sondieren

## Hashfunktion beim quadratischen Sondieren

$h(k, i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2)$ **mod** $m$ (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.



# Quadratisches Sondieren

## Hashfunktion beim quadratischen Sondieren

$h(k, i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2)$ **mod** $m$ (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).

$m!$ Permutationen

# Quadratisches Sondieren

## Hashfunktion beim quadratischen Sondieren

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

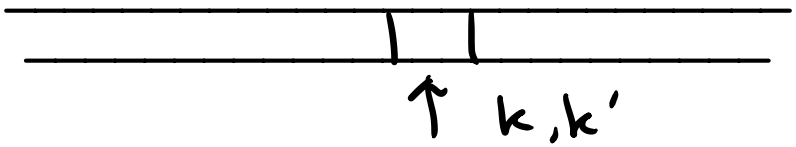
- $h'$ ist eine übliche Hashfunktion, $c_1$, $c_2 \neq 0$ Konstanten.

- 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.

# Quadratisches Sondieren




## Hashfunktion beim quadratischen Sondieren

$h(k,i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2)$ **mod** $m$ (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

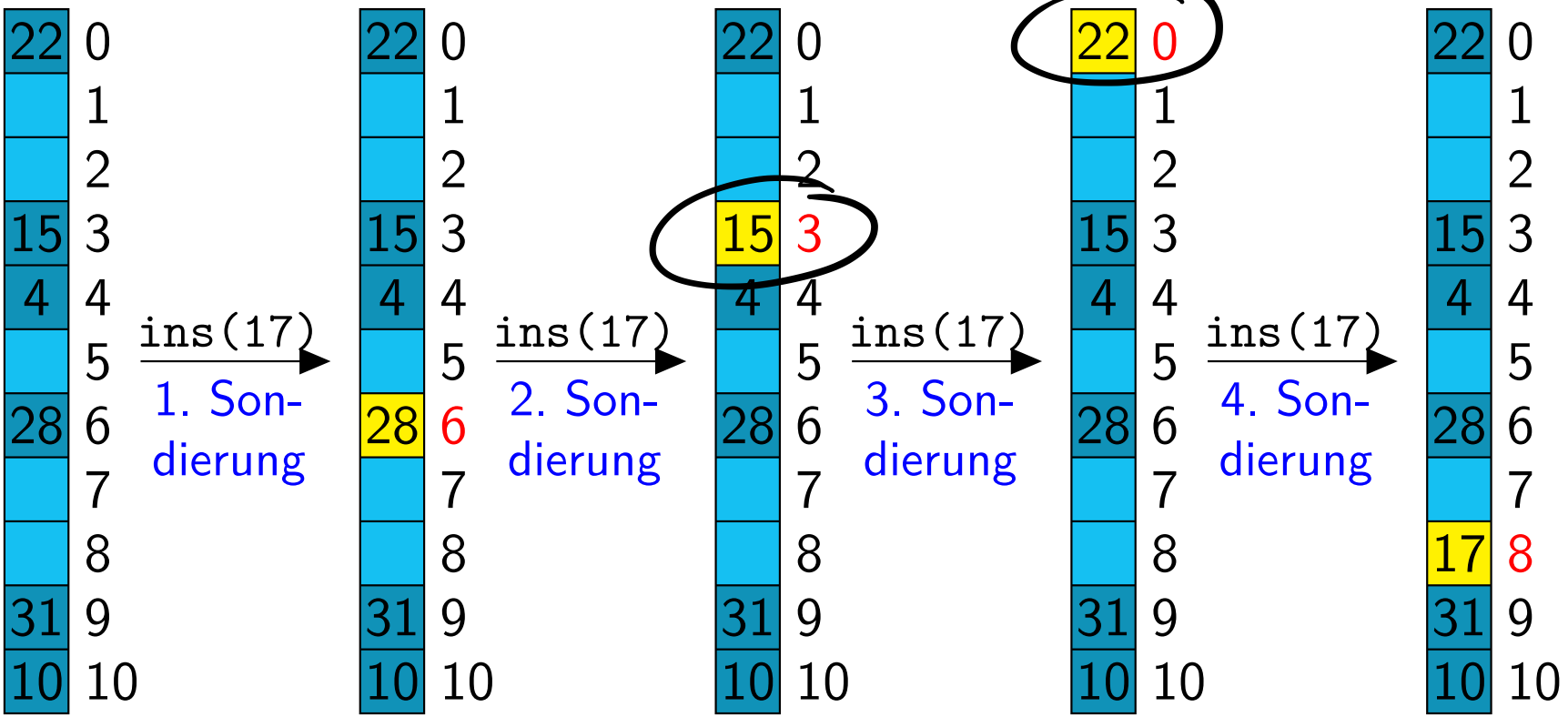
$h(k, i) = (h_1(k) + i \cdot h_2(k))$ **mod** $m$ (für $i < m$).

- $h_1$, $h_2$ sind übliche Hashfunktionen.

$+ i$

$+ i + 3i^2$

# Doppeltes Hashing: Beispiel




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

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

$$h_1(k) = k \bmod 11$$

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

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

insert (17):

$$h(17,0) = \underbrace{(17 \bmod 11)}_{=6} \bmod 11 = 6$$

$$h(17,1) = (6 + 1 \cdot (1 + \underbrace{17 \bmod 10}_{7})) \bmod 11$$

$$14 \bmod 11 = 3$$

$$h(17,2) = (6 + 2 \cdot (\underbrace{1 + 17 \bmod 10}_{8})) \bmod 11$$

$$= 22 \bmod 11$$

# Doppeltes Hashing

## Hashfunktion beim doppelten Hashing

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

- $h_1$, $h_2$ sind übliche Hashfunktionen.

# Doppeltes Hashing

## Hashfunktion beim doppelten Hashing

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

- $h_1$, $h_2$ sind übliche Hashfunktionen.

- Die Verschiebung der nachfolgende Sondierungen ist von $h_2(k)$ abhängig.

# Doppeltes Hashing

## Hashfunktion beim doppelten Hashing

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

- $h_1$, $h_2$ sind übliche Hashfunktionen.

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

$h_1(k) \mod m$

$(h_1(k) + h_2(k)) \mod m$

$(h_1(k) + 2h_2(k)) \mod m$

$k, k'$

$h_1(k) = h_1(k')$

immerhin

$h_2(k) \neq h_2(k')$

# Doppeltes Hashing

## Hashfunktion beim doppelten Hashing

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

- $h_1$, $h_2$ sind übliche Hashfunktionen.

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

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

# Doppeltes Hashing

## Hashfunktion beim doppelten Hashing

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

- $h_1$, $h_2$ sind übliche Hashfunktionen.

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

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

$\Rightarrow$ Approximiert das gleichverteilte Hashing.

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

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

$\Rightarrow$ Approximiert das gleichverteilte Hashing.

- Sind $h_2$ und $m$ relativ prim, wird die gesamte Hashtabelle abgesucht.

# Doppeltes Hashing

## Hashfunktion beim doppelten Hashing

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

- $h_1$, $h_2$ sind übliche Hashfunktionen.

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

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

$\Rightarrow$ 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.

# Doppeltes Hashing

## Hashfunktion beim doppelten Hashing

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

- $h_1$, $h_2$ sind übliche Hashfunktionen.

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

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

$\Rightarrow$ 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.

# Doppeltes Hashing

## Hashfunktion beim doppelten Hashing

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

- $h_1$, $h_2$ sind übliche Hashfunktionen.

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

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

$\Rightarrow$ 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.

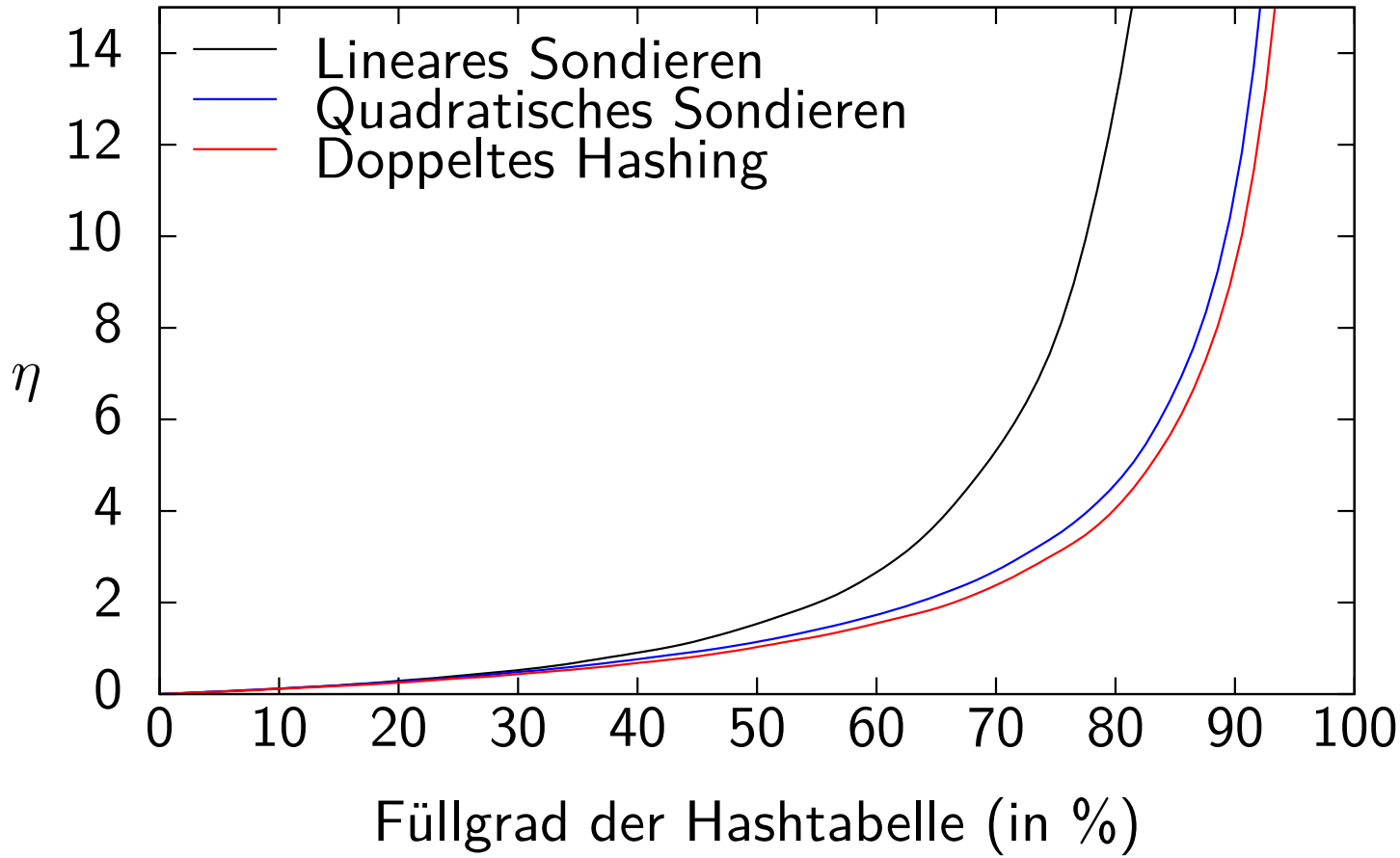
$\Rightarrow$ Daher können $m^2$ verschiedene Permutationen erzeugt werden.

# Praktische Effizienz von Doppeltem Hashing

- Hashtabelle mit 538 051 Einträgen (Endfüllgrad 99,95%)

# Praktische Effizienz von Doppeltem Hashing

- Hashtabelle mit 538 051 Einträgen (Endfüllgrad 99,95%)
- *Mittlere* Anzahl Kollisionen $\eta$ pro Einfügen in die Hashtabelle:



# Effizienz der offenen Adressierung

# Effizienz der offenen Adressierung

Unter der Annahme von gleichverteiltem Hashing gilt:

# 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.

# 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.

# 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.

# 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.

# 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.

# 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.

# 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 (I)

## Annahmen

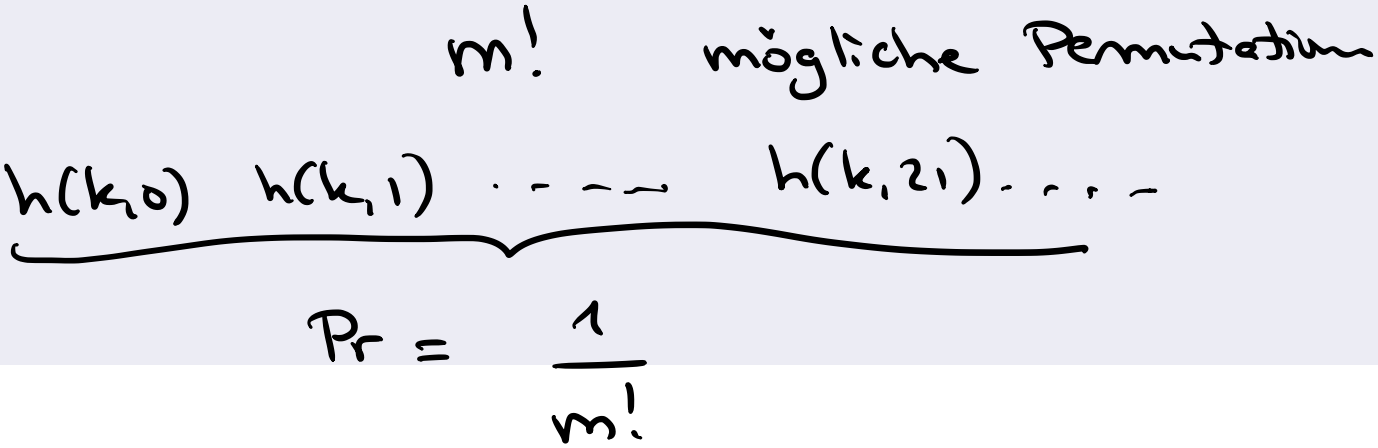
- Betrachte eine zufällig erzeugte Sondierungssequenz für Schlüssel $k$.

$h(k,0)$ $h(k,1)$ $h(k,2)$ .....

# Analyse der erfolglosen Suche (I)

## 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.



# Analyse der erfolglosen Suche (I)

## 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 $k'$, $k \neq k'$.

# Analyse der erfolglosen Suche (I)

## 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 Berechnung von Hashwerten in $O(1)$ liegt.

$h(k,i) \in \mathcal{O}(1)$

# Analyse der erfolglosen Suche (II)

## 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.

Erfolglose Suche

Suche für k ist erfolglos, d.h.

$T[h(k,0)]$, $T[h(k,1)]$, ....., $T[h(k,i-1)]$

besetzt besetzt besetzt

$T[h(k,i)] = \text{NULL}$ ← erster freier Platz

Also: erwartete Anzahl der Sondierungen für erfolglose Suche nach k = 1 + erwartete Anzahl der belegte Positionen bis erster freier Platz

Lemma sei $h: U \times \{0,\dots,m-1\} \to \{0,\dots,m-1\}$ und k so, daß $h(k,0),\dots,h(k,m-1)$ eine zufällige Permutation von $\{0,\dots,m-1\}$ ist

Seien n Positionen in T schon besetzt.

$n < m$.

Dann: $1 + \mathbb{E}[\min\{i \in \mathbb{N} \mid h(k,i) \text{ ist frei}\}]$

$\leq \frac{m+1}{m-n+1} \in O\left(\frac{1}{1-\alpha}\right)$

**Beweis:** $X = \min \{ i \in \mathbb{N} \mid h(k,i) \text{ ist frei} \}$

Ergebnis $X \geq i$ gdw. $h(k,0), \ldots, h(k,i-1)$ sind alle belegt. Also

$$\underbrace{\{h(k,0), \ldots, h(k,i-1)\}}_{\text{zufällige } i\text{-elementige Teilmenge von } \{0,\ldots,m-1\}} \subseteq \underbrace{\{ j \mid h(k,j) \text{ ist belegt} \}}_{\text{diese Menge enthält } n \text{ Elemente}}$$

$$\Pr\{X \geq i\} = \frac{\text{Anzahl der Möglichkeiten } i \text{ Positionen aus } n \text{ belegte Positionen zu wählen}}{\text{Anzahl der M'keiten } i \text{ Positionen aus alle } m \text{ verfügbare Positionen zu wählen}}$$

$$\Pr\{X \geq i\} = \frac{\binom{n}{i}}{\binom{m}{i}} = \frac{\frac{n!}{i!(n-i)!}}{\frac{m!}{i!(m-i)!}} = \frac{n!}{m!} \cdot \frac{(m-i)!}{(n-i)!}$$

$$E(X) = \sum_{i=1}^{\infty} \Pr\{X \geq i\} = \sum_{i=1}^{n} \Pr\{X \geq i\}$$

(* $\Pr(X \geq n+1) = 0$ da $n$ Positionen belegt sind *)

$$\sum_{i=1}^{n} \Pr\{X \geq i\} = \frac{n!}{m!} \sum_{i=1}^{n} \frac{(m-i)!}{(n-i)!}$$

$$= \quad (\text{* Übungsaufgabe *})$$

$$\frac{n}{m-n+1} = \mathbb{E}(X)$$

$$1 + E(X) = \frac{m-n+1}{m-n+1} + \frac{n}{m-n+1} = \frac{m+1}{m-n+1}$$

$$\in \mathcal{O}\left(\frac{m}{m-n}\right) =$$

$$\mathcal{O}\left(\frac{1}{1-\frac{n}{m}}\right) = \boxed{\mathcal{O}\left(\frac{1}{1-\alpha}\right)}$$

# Analyse der erfolglosen Suche (II)

## 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 ist:

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

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

# Analyse der erfolglosen Suche (II)

## 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 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 $1 + E[X]$.

# Analyse der erfolglosen Suche (II)

## 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 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

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

Beweis: in der Vorlesung.

Nachtrag:

Sei $k_1, k_2, \ldots, k_n$ die Folge der eingefügten Schlüsseln

average-case Komplexität der erfolgreichen Suche nach Schlüsseln $k_{j+1}$

$$= \mathbb{E}\left[\begin{array}{l}\text{Anzahl der benötigten Sondierungen} \\ \text{beim Einfügen von } k_{j+1} \text{ in } T\end{array}\right]$$

$$= \mathbb{E}\left[\min\left\{ i \in \mathbb{N} \;\middle|\; h(k_{j+1}, i) \text{ ist frei und } T \text{ enthält schon } j \text{ Schlüssel } k_1, \ldots, k_j \right\}\right]$$

$\leq$ (* Anteil der freien Positionen ist $\frac{m-j}{m}$ bzgl. alle Positionen *)

$$\frac{m}{m-j}.$$

Also: erwartete Zeit für Schlüssel $k_{j+1}$

ist $\leq \frac{m}{m-j}$

Wir betrachten jetzt die mittlere Zeit für <u>alle</u> n Schlüssel die sich in T befinden:

$$\frac{1}{n} \cdot \sum_{i=0}^{n-1} \mathbb{E}\left[\text{Anzahl Sondierungen beim Einfügen von } k_{i+1}\right]$$

$$\leq \frac{1}{n} \cdot \sum_{i=0}^{n-1} \frac{m}{m-i}$$

$$= \frac{m}{n} \cdot \sum_{i=0}^{n-1} \frac{1}{m-i}$$

$\frac{1}{m} + \ldots + \frac{1}{m-(n-1)}$

$$= \frac{1}{\alpha} \cdot \sum_{k=m-n+1}^{m} \frac{1}{k} \qquad (* \ \alpha = \frac{n}{m} \ *)$$

$$= \frac{1}{\alpha} \cdot \left(H_m - H_{m-n}\right) \qquad (* \ H_n = \sum_{i=1}^{n} \frac{1}{k} \ *)$$

$$\in \mathcal{O}\left(\frac{1}{\alpha}\left(\ln(m) - \ln(m-n)\right)\right) \qquad (* \ H_n \in \mathcal{O}(\ln(n)) \ *)$$

$$\in \mathcal{O}\left(\frac{1}{\alpha} \cdot \ln\left(\frac{m}{m-n}\right)\right) \qquad (* \ \alpha = \frac{n}{m} \ *)$$

$$\in \mathcal{O}\left(\frac{1}{\alpha} \cdot \ln\left(\frac{1}{1-\alpha}\right)\right)$$

# Analyse der erfolglosen Suche (II)

## 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 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

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

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

# Analyse der erfolgeichen Suche (I)

- Sei Schlüssel $k_i$ der $i$-te eingefügte Schlüssel in der Hashtabelle.

# Analyse der erfolgeichen Suche (I)

- Sei Schlüssel $k_i$ der $i$-te eingefügte Schlüssel in der Hashtabelle.
- Betrachte eine erfolgreiche Suche für Schlüssel $k_{i+1}$.

# Analyse der erfolgeichen 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 $k_i$.

# Analyse der erfolgeichen 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.

# Analyse der erfolgeichen 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:

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

# Analyse der erfolgeichen 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.

# Nächste Vorlesung

## Nächste Vorlesung

Freitag 15. Juni, 13:15 (Hörsaal H01). Bis dann!