

SS18

AKKTA: Hashverfahren

0. Kapitel: Einführung, Beispiele

Hashverfahren

Martin Dietzfelbinger

Mai 2018

0.1 Grundbegriffe

Gegeben:

Universum U (auch ohne Ordnung) und Wertebereich R .

0.1 Grundbegriffe

Gegeben:

Universum U (auch ohne Ordnung) und Wertebereich R .

Hashverfahren (oder **Schlüsseltransformationsverfahren**)
implementieren **Wörterbuch**

0.1 Grundbegriffe

Gegeben:

Universum U (auch ohne Ordnung) und Wertebereich R .

Hashverfahren (oder **Schlüsseltransformationsverfahren**)
implementieren **Wörterbuch** oder **dynamische Abbildung**

0.1 Grundbegriffe

Gegeben:

Universum U (auch ohne Ordnung) und Wertebereich R .

Hashverfahren (oder **Schlüsseltransformationsverfahren**)

implementieren **Wörterbuch** oder **dynamische Abbildung**

$f : S \rightarrow R$, mit $S \subseteq U$ endlich.

0.1 Grundbegriffe

Gegeben:

Universum U (auch ohne Ordnung) und Wertebereich R .

Hashverfahren (oder **Schlüsseltransformationsverfahren**)

implementieren **Wörterbuch** oder **dynamische Abbildung**

$f : S \rightarrow R$, mit $S \subseteq U$ endlich.

Ziel:

0.1 Grundbegriffe

Gegeben:

Universum U (auch ohne Ordnung) und Wertebereich R .

Hashverfahren (oder **Schlüsseltransformationsverfahren**)
implementieren **Wörterbuch** oder **dynamische Abbildung**

$f : S \rightarrow R$, mit $S \subseteq U$ endlich.

Ziel:

Zeit **$O(1)$** pro Operation

0.1 Grundbegriffe

Gegeben:

Universum U (auch ohne Ordnung) und Wertebereich R .

Hashverfahren (oder **Schlüsseltransformationsverfahren**)
implementieren **Wörterbuch** oder **dynamische Abbildung**

$f : S \rightarrow R$, mit $S \subseteq U$ endlich.

Ziel:

Zeit **$O(1)$** pro Operation
„im Durchschnitt“

0.1 Grundbegriffe

Gegeben:

Universum U (auch ohne Ordnung) und Wertebereich R .

Hashverfahren (oder **Schlüsseltransformationsverfahren**)
implementieren **Wörterbuch** oder **dynamische Abbildung**

$$f : S \rightarrow R, \text{ mit } S \subseteq U \text{ endlich.}$$

Ziel:

Zeit $O(1)$ pro Operation
„im Durchschnitt“
unabhängig vom Umfang der Datenstruktur!

0.1 Grundbegriffe

Gegeben:

Universum U (auch ohne Ordnung) und Wertebereich R .

Hashverfahren (oder **Schlüsseltransformationsverfahren**)
implementieren **Wörterbuch** oder **dynamische Abbildung**

$$f : S \rightarrow R, \text{ mit } S \subseteq U \text{ endlich.}$$

Ziel:

Zeit $O(1)$ pro Operation
„im Durchschnitt“
unabhängig vom Umfang der Datenstruktur!

Gegensatz **Suchbäume:**

0.1 Grundbegriffe

Gegeben:

Universum U (auch ohne Ordnung) und Wertebereich R .

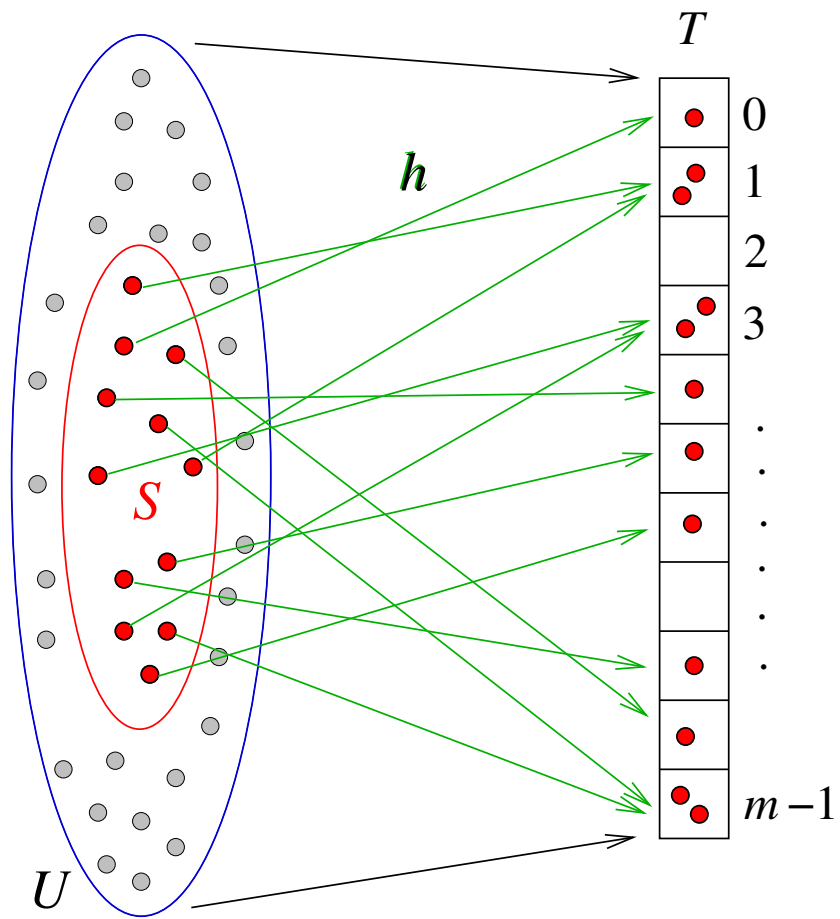
Hashverfahren (oder **Schlüsseltransformationsverfahren**)
implementieren **Wörterbuch** oder **dynamische Abbildung**

$$f : S \rightarrow R, \text{ mit } S \subseteq U \text{ endlich.}$$

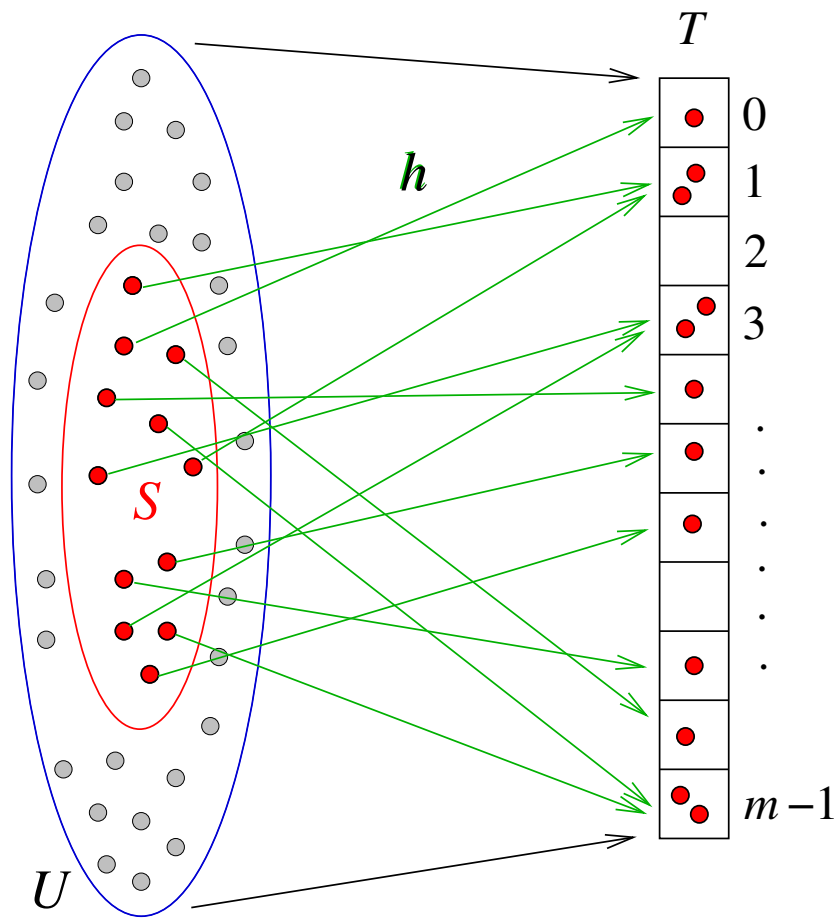
Ziel:

Zeit $O(1)$ pro Operation
„im Durchschnitt“
unabhängig vom Umfang der Datenstruktur!

Gegensatz **Suchbäume**: Zeit $O(\log(n))$ pro Operation.

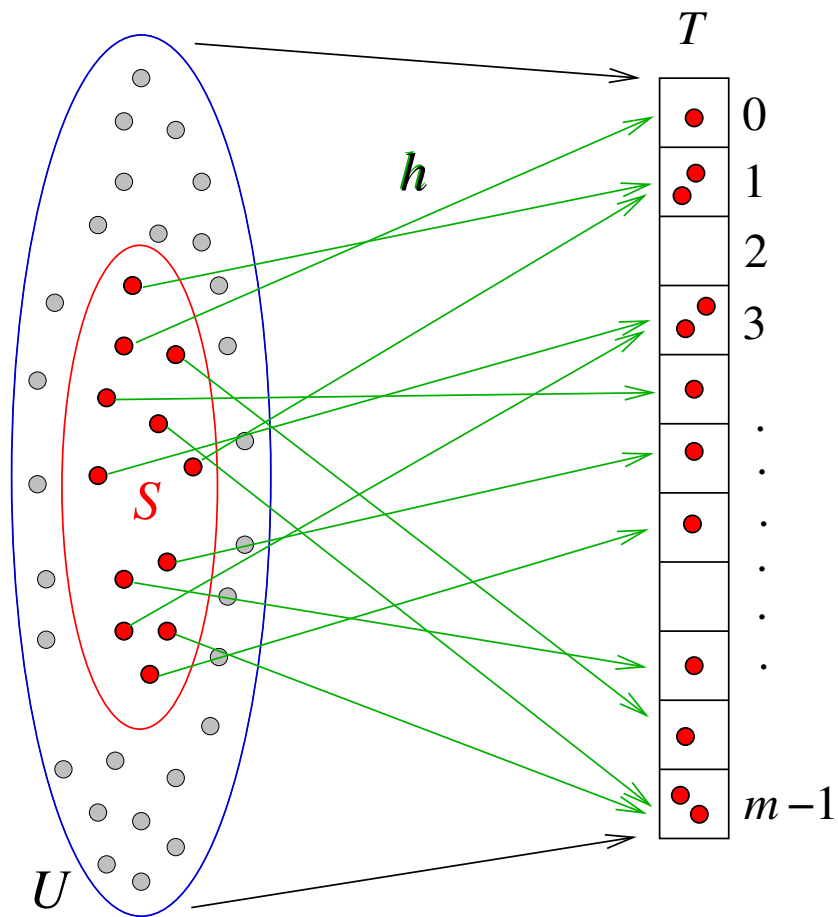


* to hash (*engl.*): kleinhacken, kleinschneiden



Hashfunktion*
 $h: U \rightarrow [m]$

* to hash (*engl.*): kleinhacken, kleinschneiden

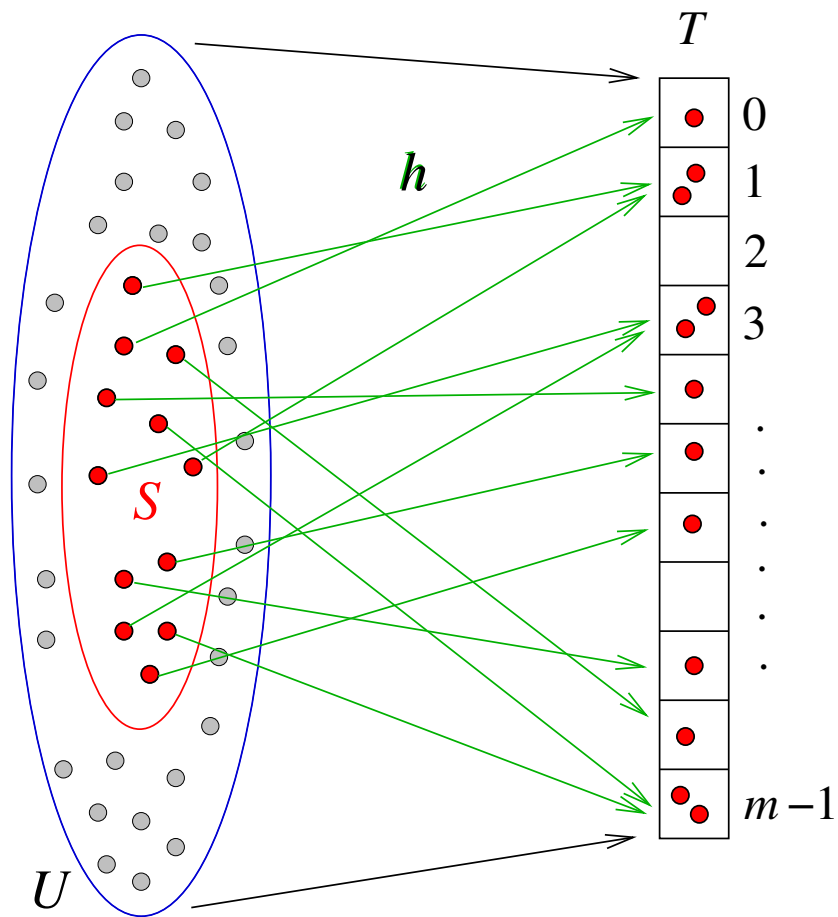


Hashfunktion*

$$h: U \rightarrow [m]$$

U : Universum der Schlüssel

* to hash (*engl.*): kleinhacken, kleinschneiden



Hashfunktion*

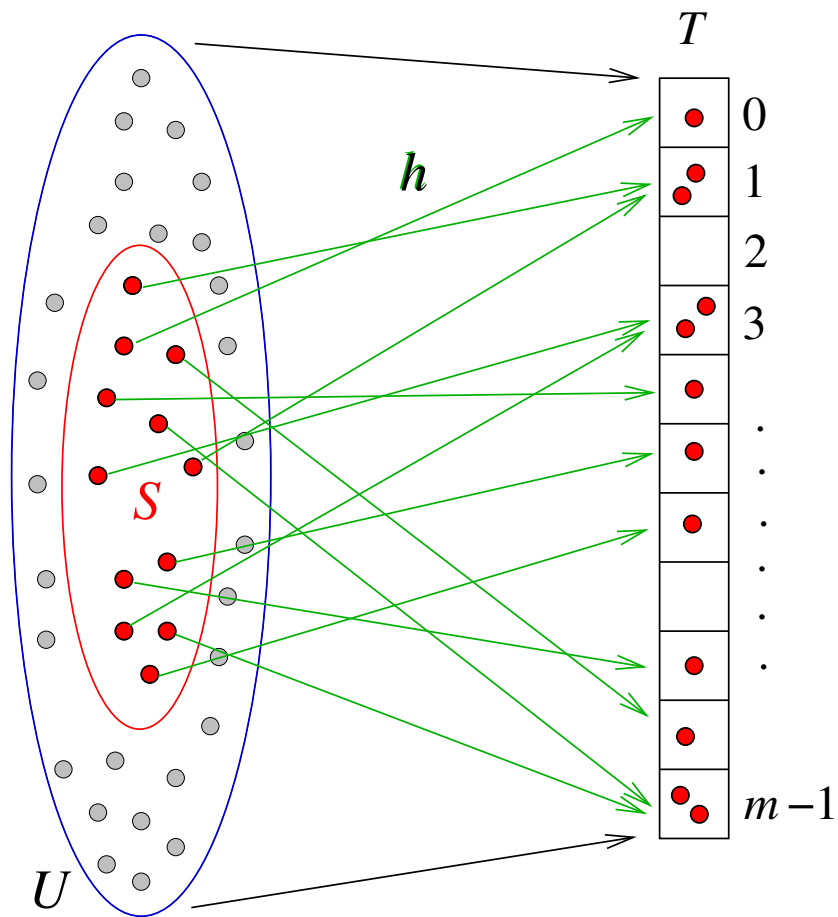
$$h: U \rightarrow [m]$$

U : Universum der Schlüssel

$$[m] = \{0, \dots, m - 1\} :$$

Indexbereich für Tabelle T

* to hash (*engl.*): kleinhacken, kleinschneiden



Hashfunktion*

$$h: U \rightarrow [m]$$

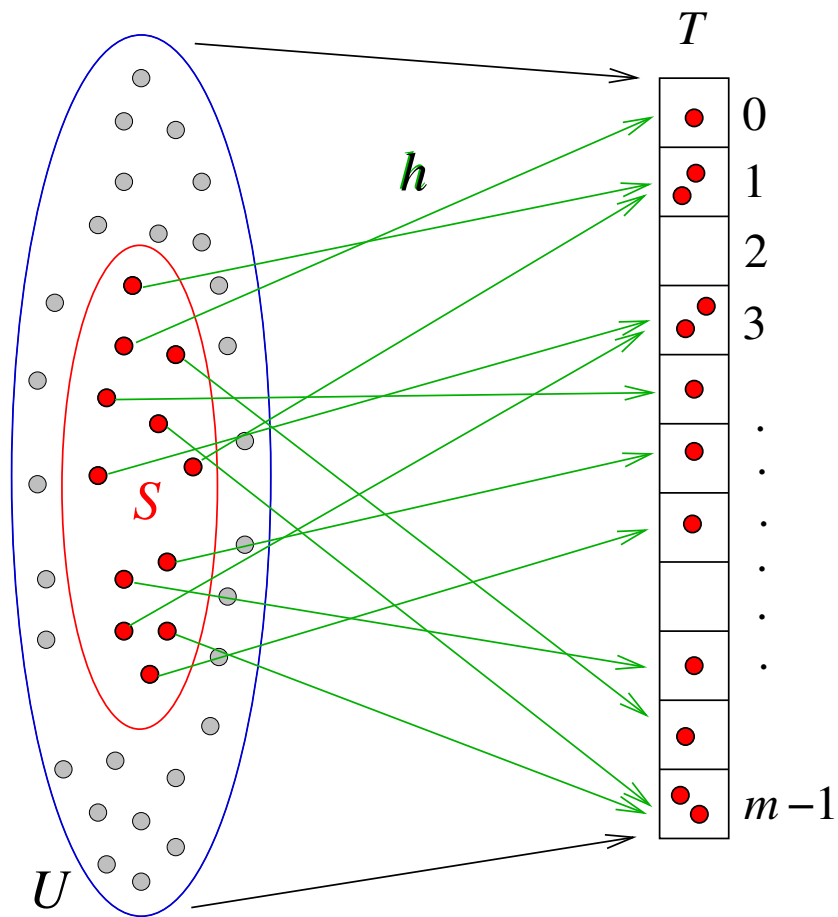
U : Universum der Schlüssel

$$[m] = \{0, \dots, m - 1\} :$$

Indexbereich für Tabelle T

$$S \subseteq U$$

* to hash (*engl.*): kleinhacken, kleinschneiden



Hashfunktion*

$$h: U \rightarrow [m]$$

U : Universum der Schlüssel

$$[m] = \{0, \dots, m - 1\} :$$

Indexbereich für Tabelle T

$$S \subseteq U, |S| = n$$

* to hash (*engl.*): kleinhacken, kleinschneiden

Grundansatz:

Grundansatz:

Speicherung von $f: S \rightarrow R$ in m Tabelle $T[0..m-1]$.

Grundansatz:

Speicherung von $f: S \rightarrow R$ in m Tabelle $T[0..m-1]$.

Aus Schlüssel $x \in U$ wird ein Index $h(x) \in [m]$ **berechnet**.

Grundansatz:

Speicherung von $f: S \rightarrow R$ in m Tabelle $T[0..m-1]$.

Aus Schlüssel $x \in U$ wird ein Index $h(x) \in [m]$ **berechnet**.

„ x wird in $h(x)$ umgewandelt/**transformiert**.“

Grundansatz:

Speicherung von $f: S \rightarrow R$ in m Tabelle $T[0..m-1]$.

Aus Schlüssel $x \in U$ wird ein Index $h(x) \in [m]$ **berechnet**.

„ x wird in $h(x)$ umgewandelt/**transformiert**.“

Idealvorstellung:

Speichere Paar $(x, f(x))$ in Zelle $T[h(x)]$.

$$h(c_1 \dots c_r) = \text{num}(c_3) \bmod 13$$

j	$(x, f(x))$ mit $h(x) = j$
0	(Januar, 31); (Juni, 30)
1	(Februar, 28)
2	(September, 30)
3	
4	(Maerz, 31); (April, 30)
5	
6	(August, 31); (Oktober, 31)
7	
8	(Mai, 31); (November, 30)
9	
10	
11	(Juli, 31)
12	(Dezember, 31)

Kollisionen bei $j = 0, 3, 6, 8$.

Definition 0.1.1

Eine Funktion $h: U \rightarrow [m]$ heißt eine **Hashfunktion**.

Definition 0.1.1

Eine Funktion $h: U \rightarrow [m]$ heißt eine **Hashfunktion**.

Wenn $S \subseteq U$ gegeben ist (oder ein $f: S \rightarrow R$), verteilt h die Schlüssel in S auf $[m]$.

Definition 0.1.1

Eine Funktion $h: U \rightarrow [m]$ heißt eine **Hashfunktion**.

Wenn $S \subseteq U$ gegeben ist (oder ein $f: S \rightarrow R$), verteilt h die Schlüssel in S auf $[m]$.

$$B_j := \{x \in S \mid h(x) = j\}.$$

Definition 0.1.1

Eine Funktion $h: U \rightarrow [m]$ heißt eine **Hashfunktion**.

Wenn $S \subseteq U$ gegeben ist (oder ein $f: S \rightarrow R$), verteilt h die Schlüssel in S auf $[m]$.

$B_j := \{x \in S \mid h(x) = j\}$. „Behälter“ („bucket“)

Definition 0.1.1

Eine Funktion $h: U \rightarrow [m]$ heißt eine **Hashfunktion**.

Wenn $S \subseteq U$ gegeben ist (oder ein $f: S \rightarrow R$), verteilt h die Schlüssel in S auf $[m]$.

$B_j := \{x \in S \mid h(x) = j\}$. „Behälter“ („bucket“)

Idealfall: Die Einschränkung $h \upharpoonright S$ ist **injektiv**, d. h. für jedes $j \in [m]$ existiert höchstens ein $x \in S$ mit $h(x) = j$.

Definition 0.1.1

Eine Funktion $h: U \rightarrow [m]$ heißt eine **Hashfunktion**.

Wenn $S \subseteq U$ gegeben ist (oder ein $f: S \rightarrow R$), verteilt h die Schlüssel in S auf $[m]$.

$B_j := \{x \in S \mid h(x) = j\}$. „Behälter“ („bucket“)

Idealfall: Die Einschränkung $h \upharpoonright S$ ist **injektiv**, d. h. für jedes $j \in [m]$ existiert höchstens ein $x \in S$ mit $h(x) = j$.

Beispiel: Auf

$S_1 = \{\text{Februar}^{(1)}, \text{Maerz}^{(4)}, \text{Mai}^{(8)}, \text{Juni}^{(0)}, \text{Juli}^{(11)}, \text{Dezember}^{(12)}\}$

ist obiges h injektiv,

Definition 0.1.1

Eine Funktion $h: U \rightarrow [m]$ heißt eine **Hashfunktion**.

Wenn $S \subseteq U$ gegeben ist (oder ein $f: S \rightarrow R$), verteilt h die Schlüssel in S auf $[m]$.

$B_j := \{x \in S \mid h(x) = j\}$. „Behälter“ („bucket“)

Idealfall: Die Einschränkung $h \upharpoonright S$ ist **injektiv**, d. h. für jedes $j \in [m]$ existiert höchstens ein $x \in S$ mit $h(x) = j$.

Beispiel: Auf

$S_1 = \{\text{Februar}^{(1)}, \text{Maerz}^{(4)}, \text{Mai}^{(8)}, \text{Juni}^{(0)}, \text{Juli}^{(11)}, \text{Dezember}^{(12)}\}$

ist obiges h injektiv,

nicht aber auf $S_2 = \{\text{Januar}^{(0)}, \text{Mai}^{(8)}, \text{Juli}^{(11)}, \text{November}^{(8)}\}$.

Sprechweise: h **perfekt für** S $:\Leftrightarrow h \upharpoonright S$ injektiv.

Sprechweise: h **perfekt für S** $:\Leftrightarrow h \upharpoonright S$ injektiv.

Falls h perfekt für S :

Speichere x bzw. (x, r) in Tabellenplatz $T[h(x)]$, für $x \in S$.

Sprechweise: h **perfekt für S** $:\Leftrightarrow h \upharpoonright S$ injektiv.

Falls h perfekt für S :

Speichere x bzw. (x, r) in Tabellenplatz $T[h(x)]$, für $x \in S$.

→ Konstante Zugriffszeit.

Sprechweise: h **perfekt für** S $:\Leftrightarrow h \upharpoonright S$ injektiv.

Falls h perfekt für S :

Speichere x bzw. (x, r) in Tabellenplatz $T[h(x)]$, für $x \in S$.

→ Konstante Zugriffszeit.

Kollision: $x, y \in S, x \neq y, h(x) = h(y)$.

Sprechweise: h **perfekt für** S $:\Leftrightarrow h \upharpoonright S$ injektiv.

Falls h perfekt für S :

Speichere x bzw. (x, r) in Tabellenplatz $T[h(x)]$, für $x \in S$.

→ Konstante Zugriffszeit.

Kollision: $x, y \in S, x \neq y, h(x) = h(y)$.

Bei zufälligen Hashwerten sind Kollisionen „praktisch unvermeidlich“.

Fall: h „rein zufällig“, $|S| = n$, $S = \{x_1, \dots, x_n\}$. D. h.:

Fall: h „rein zufällig“, $|S| = n$, $S = \{x_1, \dots, x_n\}$. D. h.:

Uniformitätsannahme (UF*):

Werte $h(x_1), \dots, h(x_n)$ in $[m]^n$ sind „rein zufällig“, d. h.:
jeder der m^n Vektoren $(j_1, \dots, j_n) \in [m]^n$
kommt mit derselben Wahrscheinlichkeit $1/m^n$
als $(h(x_1), \dots, h(x_n))$ vor.

Mit (UF^*) :

Mit (UF^*) : $\Pr(\text{keine Kollision}) =$

Mit (UF*): $\Pr(\text{keine Kollision}) =$

$$= \Pr(h \text{ ist injektiv auf } \{x_1, \dots, x_n\}) =$$

Mit (UF*): $\Pr(\text{keine Kollision}) =$

$$= \Pr(h \text{ ist injektiv auf } \{x_1, \dots, x_n\}) =$$

$$= \frac{m}{m}$$

Mit (UF*): $\Pr(\text{keine Kollision}) =$

$$= \Pr(h \text{ ist injektiv auf } \{x_1, \dots, x_n\}) =$$

$$= \frac{m}{m} \cdot \frac{m-1}{m}$$

Mit (UF*): $\Pr(\text{keine Kollision}) =$

$$= \Pr(h \text{ ist injektiv auf } \{x_1, \dots, x_n\}) =$$

$$= \frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m}$$

Mit (UF*): $\Pr(\text{keine Kollision}) =$

$$\begin{aligned} &= \Pr(h \text{ ist injektiv auf } \{x_1, \dots, x_n\}) = \\ &= \frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdot \dots \cdot \frac{m-n+1}{m} = \dots \end{aligned}$$

Mit (UF*): $\Pr(\text{keine Kollision}) =$

$$\begin{aligned} &= \Pr(h \text{ ist injektiv auf } \{x_1, \dots, x_n\}) = \\ &= \frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdot \dots \cdot \frac{m-n+1}{m} = \dots \end{aligned}$$

(Für $i = 1, \dots, n$ ist die Wahrscheinlichkeit, dass $h(x_i)$ in $[m] - \{h(x_1), \dots, h(x_{i-1})\}$ liegt, genau $\frac{m-i+1}{m}$.)

Mit (UF*): $\Pr(\text{keine Kollision}) =$

$$\begin{aligned} &= \Pr(h \text{ ist injektiv auf } \{x_1, \dots, x_n\}) = \\ &= \frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdot \dots \cdot \frac{m-n+1}{m} = \dots \end{aligned}$$

(Für $i = 1, \dots, n$ ist die Wahrscheinlichkeit, dass $h(x_i)$ in $[m] - \{h(x_1), \dots, h(x_{i-1})\}$ liegt, genau $\frac{m-i+1}{m}$.)

$$\dots = 1 \cdot \left(1 - \frac{1}{m}\right) \cdot \left(1 - \frac{2}{m}\right) \cdot \dots \cdot \left(1 - \frac{n-1}{m}\right)$$

Mit (UF*): $\Pr(\text{keine Kollision}) =$

$$\begin{aligned} &= \Pr(h \text{ ist injektiv auf } \{x_1, \dots, x_n\}) = \\ &= \frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdot \dots \cdot \frac{m-n+1}{m} = \dots \end{aligned}$$

(Für $i = 1, \dots, n$ ist die Wahrscheinlichkeit, dass $h(x_i)$ in $[m] - \{h(x_1), \dots, h(x_{i-1})\}$ liegt, genau $\frac{m-i+1}{m}$.)

$$\begin{aligned} \dots &= 1 \cdot \left(1 - \frac{1}{m}\right) \cdot \left(1 - \frac{2}{m}\right) \cdot \dots \cdot \left(1 - \frac{n-1}{m}\right) \\ &< e^0 \cdot e^{-\frac{1}{m}} \cdot e^{-\frac{2}{m}} \cdot \dots \cdot e^{-\frac{n-1}{m}} \end{aligned}$$

Mit (UF*): $\Pr(\text{keine Kollision}) =$

$$\begin{aligned} &= \Pr(h \text{ ist injektiv auf } \{x_1, \dots, x_n\}) = \\ &= \frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdot \dots \cdot \frac{m-n+1}{m} = \dots \end{aligned}$$

(Für $i = 1, \dots, n$ ist die Wahrscheinlichkeit, dass $h(x_i)$ in $[m] - \{h(x_1), \dots, h(x_{i-1})\}$ liegt, genau $\frac{m-i+1}{m}$.)

$$\begin{aligned} \dots &= 1 \cdot \left(1 - \frac{1}{m}\right) \cdot \left(1 - \frac{2}{m}\right) \cdot \dots \cdot \left(1 - \frac{n-1}{m}\right) \\ &< e^0 \cdot e^{-\frac{1}{m}} \cdot e^{-\frac{2}{m}} \cdot \dots \cdot e^{-\frac{n-1}{m}} = e^{-\frac{n(n-1)}{2m}}. \end{aligned}$$

Mit (UF*): $\Pr(\text{keine Kollision}) =$

$$\begin{aligned} &= \Pr(h \text{ ist injektiv auf } \{x_1, \dots, x_n\}) = \\ &= \frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdot \dots \cdot \frac{m-n+1}{m} = \dots \end{aligned}$$

(Für $i = 1, \dots, n$ ist die Wahrscheinlichkeit, dass $h(x_i)$ in $[m] - \{h(x_1), \dots, h(x_{i-1})\}$ liegt, genau $\frac{m-i+1}{m}$.)

$$\begin{aligned} \dots &= 1 \cdot \left(1 - \frac{1}{m}\right) \cdot \left(1 - \frac{2}{m}\right) \cdot \dots \cdot \left(1 - \frac{n-1}{m}\right) \\ &< e^0 \cdot e^{-\frac{1}{m}} \cdot e^{-\frac{2}{m}} \cdot \dots \cdot e^{-\frac{n-1}{m}} = e^{-\frac{n(n-1)}{2m}}. \end{aligned}$$

(denn für $x \neq 0$ ist $1 + x < e^x$),

Mit (UF*): $\Pr(\text{keine Kollision}) =$

$$\begin{aligned} &= \Pr(h \text{ ist injektiv auf } \{x_1, \dots, x_n\}) = \\ &= \frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdot \dots \cdot \frac{m-n+1}{m} = \dots \end{aligned}$$

(Für $i = 1, \dots, n$ ist die Wahrscheinlichkeit, dass $h(x_i)$ in $[m] - \{h(x_1), \dots, h(x_{i-1})\}$ liegt, genau $\frac{m-i+1}{m}$.)

$$\begin{aligned} \dots &= 1 \cdot \left(1 - \frac{1}{m}\right) \cdot \left(1 - \frac{2}{m}\right) \cdot \dots \cdot \left(1 - \frac{n-1}{m}\right) \\ &< e^0 \cdot e^{-\frac{1}{m}} \cdot e^{-\frac{2}{m}} \cdot \dots \cdot e^{-\frac{n-1}{m}} = e^{-\frac{n(n-1)}{2m}}. \end{aligned}$$

(denn für $x \neq 0$ ist $1 + x < e^x$), also . . .

$$\Pr(h \text{ injektiv auf } S) < e^{-\frac{n(n-1)}{2m}}.$$

“Geburtstagsparadoxon”:

$$\Pr(h \text{ injektiv auf } S) < e^{-\frac{n(n-1)}{2m}}.$$

“Geburtstagsparadoxon”:

x_1, \dots, x_n sind Leute, $h(x_1), \dots, h(x_n) \in [364]$ ihre Geburtstage, $m = 365$.

$$\Pr(h \text{ injektiv auf } S) < e^{-\frac{n(n-1)}{2m}}.$$

“Geburtstagsparadoxon”:

x_1, \dots, x_n sind Leute, $h(x_1), \dots, h(x_n) \in [364]$ ihre Geburtstage, $m = 365$.

Wenn z.B. $n = 23$ ist, ist

$$\Pr(\text{alle Geburtstage verschieden}) < e^{-23 \cdot 22 / 730} = e^{-506 / 730} < 0,5.$$

$$\Pr(h \text{ injektiv auf } S) < e^{-\frac{n(n-1)}{2m}}.$$

“Geburtstagsparadoxon”:

x_1, \dots, x_n sind Leute, $h(x_1), \dots, h(x_n) \in [364]$ ihre Geburtstage, $m = 365$.

Wenn z.B. $n = 23$ ist, ist

$$\Pr(\text{alle Geburtstage verschieden}) < e^{-23 \cdot 22 / 730} = e^{-506 / 730} < 0,5.$$

D.h.: Mit Wahrscheinlichkeit $\frac{1}{2}$ sind unter 23 Personen mit zufälligen Geburtstagen zwei, die am selben Tag Geburtstag haben!

$$\Pr(h \text{ injektiv auf } S) < e^{-\frac{n(n-1)}{2m}}.$$

Wunsch: Linearer Platzverbrauch, also

$$\Pr(h \text{ injektiv auf } S) < e^{-\frac{n(n-1)}{2m}}.$$

Wunsch: Linearer Platzverbrauch, also

„Auslastungsfaktor“ $\alpha = \frac{n}{m}$

$$\Pr(h \text{ injektiv auf } S) < e^{-\frac{n(n-1)}{2m}}.$$

Wunsch: Linearer Platzverbrauch, also

„**Auslastungsfaktor**“ $\alpha = \frac{n}{m}$

nicht kleiner als eine Konstante α_0 , z. B. $\alpha \geq \frac{1}{2}$ stets.

$$\Pr(h \text{ injektiv auf } S) < e^{-\frac{n(n-1)}{2m}}.$$

Wunsch: Linearer Platzverbrauch, also

$$\text{„Auslastungsfaktor“ } \alpha = \frac{n}{m}$$

nicht kleiner als eine Konstante α_0 , z. B. $\alpha \geq \frac{1}{2}$ stets. Dann:

$$\Pr(h \text{ injektiv auf } S) \leq e^{-\alpha_0(n-1)/2}.$$

$$\Pr(h \text{ injektiv auf } S) < e^{-\frac{n(n-1)}{2m}}.$$

Wunsch: Linearer Platzverbrauch, also

$$\text{„Auslastungsfaktor“ } \alpha = \frac{n}{m}$$

nicht kleiner als eine Konstante α_0 , z. B. $\alpha \geq \frac{1}{2}$ stets. Dann:

$$\Pr(h \text{ injektiv auf } S) \leq e^{-\alpha_0(n-1)/2}.$$

Winzig für nicht ganz kleine n !

$$\Pr(h \text{ injektiv auf } S) < e^{-\frac{n(n-1)}{2m}}.$$

Wunsch: Linearer Platzverbrauch, also

$$\text{„Auslastungsfaktor“ } \alpha = \frac{n}{m}$$

nicht kleiner als eine Konstante α_0 , z. B. $\alpha \geq \frac{1}{2}$ stets. Dann:

$$\Pr(h \text{ injektiv auf } S) \leq e^{-\alpha_0(n-1)/2}.$$

Winzig für nicht ganz kleine n ! D.h.: Falls h nicht auf S abgestimmt ist, kommen Kollisionen praktisch immer vor.

$$\Pr(h \text{ injektiv auf } S) < e^{-\frac{n(n-1)}{2m}}.$$

Wunsch: Linearer Platzverbrauch, also

$$\text{„Auslastungsfaktor“ } \alpha = \frac{n}{m}$$

nicht kleiner als eine Konstante α_0 , z. B. $\alpha \geq \frac{1}{2}$ stets. Dann:

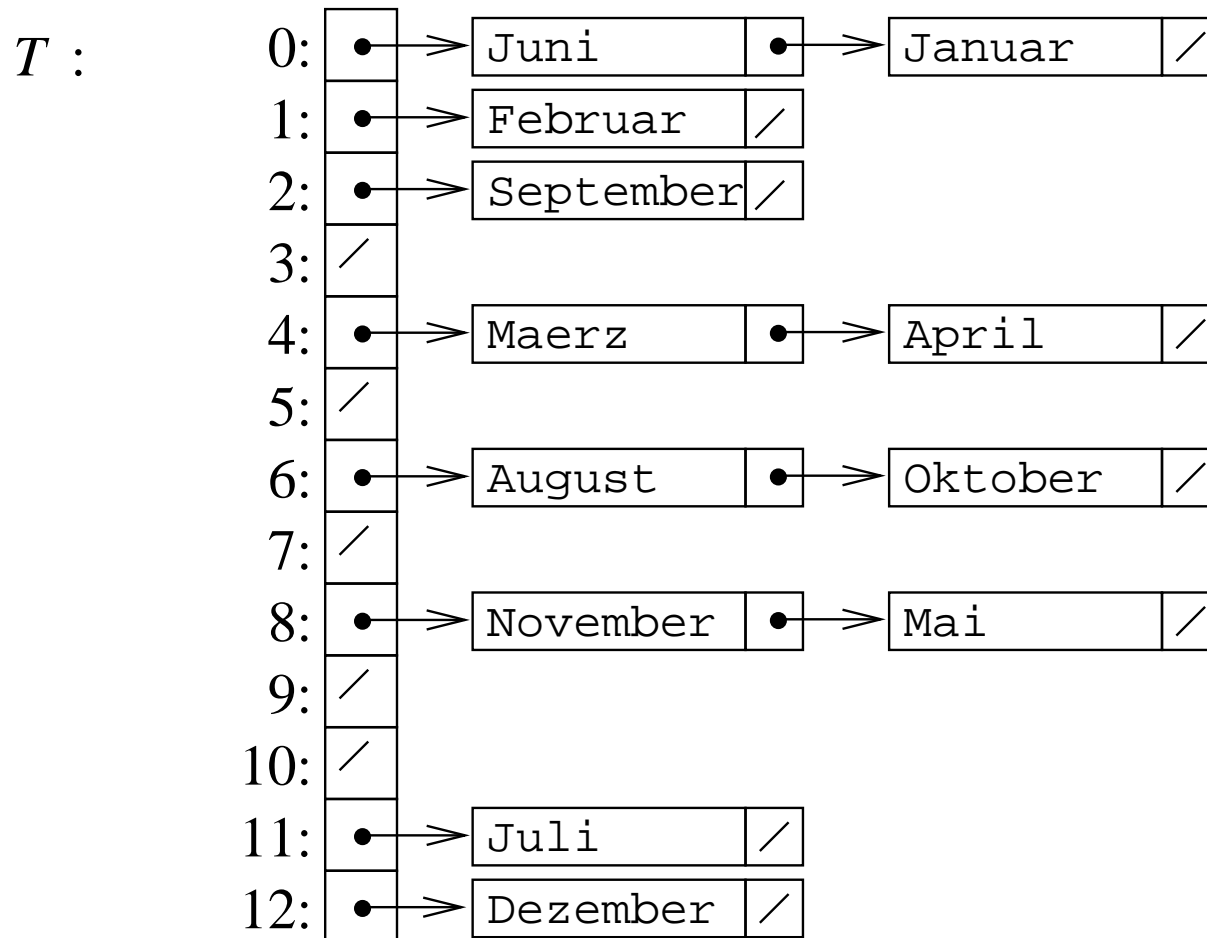
$$\Pr(h \text{ injektiv auf } S) \leq e^{-\alpha_0(n-1)/2}.$$

Winzig für nicht ganz kleine n ! D.h.: Falls h nicht auf S abgestimmt ist, kommen Kollisionen praktisch immer vor.

→ **Kollisionsbehandlung** ist notwendig.

0.2 Hashing mit verketteten Listen

Beispiel:



0.2 Hashing mit verketteten Listen

engl.: **chained hashing**

0.2 Hashing mit verketteten Listen

engl.: **chained hashing**

Auch: „**offenes Hashing**“, weil man Platz außerhalb von T benutzt.

0.2 Hashing mit verketteten Listen

engl.: **chained hashing**

Auch: „**offenes Hashing**“, weil man Platz außerhalb von T benutzt.

In $T[0..m-1]$: **Zeiger** auf die Anfänge von

0.2 Hashing mit verketteten Listen

engl.: **chained hashing**

Auch: „**offenes Hashing**“, weil man Platz außerhalb von T benutzt.

In $T[0..m-1]$: **Zeiger** auf die Anfänge
von m **einfach verketteten linearen Listen** L_0, \dots, L_{m-1} .

0.2 Hashing mit verketteten Listen

engl.: **chained hashing**

Auch: „**offenes Hashing**“, weil man Platz außerhalb von T benutzt.

In $T[0..m-1]$: **Zeiger** auf die Anfänge von m **einfach verketteten linearen Listen** L_0, \dots, L_{m-1} .

Liste L_j enthält die Einträge mit Schlüsseln x aus Behälter/Bucket

$$B_j = \{x \in S \mid h(x) = j\}.$$

Algorithmen(skizzen)

Algorithmen(skizzen)

empty(m): Erzeugt Array $T[0..m - 1]$ mit m
NULL-Zeigern/NULL-Referenzen

Algorithmen(skizzen)

empty(m): Erzeugt Array $T[0..m-1]$ mit m
NULL-Zeigern/NULL-Referenzen
und legt $h: U \rightarrow [m]$ fest.

Algorithmen(skizzen)

empty(m): Erzeugt Array $T[0..m-1]$ mit m
NULL-Zeigern/NULL-Referenzen
und legt $h: U \rightarrow [m]$ fest.

Kosten:

Algorithmen(skizzen)

empty(m): Erzeugt Array $T[0..m - 1]$ mit m
NULL-Zeigern/NULL-Referenzen
und legt $h: U \rightarrow [m]$ fest.

Kosten: $\Theta(m)$.

Algorithmen(skizzen)

empty(m): Erzeugt Array $T[0..m-1]$ mit m
NULL-Zeigern/NULL-Referenzen
und legt $h: U \rightarrow [m]$ fest.

Kosten: $\Theta(m)$.

Beachte: Man wählt h , ohne S zu kennen!

lookup(x): Berechne $j = h(x)$;

lookup(x): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;

lookup(x): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;
falls Eintrag (x, r) gefunden: **return** r ;
// **erfolgreiche Suche**

lookup(x): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;
falls Eintrag (x, r) gefunden: **return** r ;
 // **erfolgreiche Suche**
sonst: **return** „*nicht vorhanden*“
 // **erfolglose Suche**

lookup(x): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;
falls Eintrag (x, r) gefunden: **return** r ;
 // **erfolgreiche Suche**
sonst: **return** „*nicht vorhanden*“
 // **erfolglose Suche**

Kosten/Zeit:

lookup(x): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;
falls Eintrag (x, r) gefunden: **return** r ;
 // **erfolgreiche Suche**
sonst: **return** „nicht vorhanden“
 // **erfolglose Suche**

Kosten/Zeit: $O(1 + \text{Zahl der Schlüsselvergleiche „}x = y\text{“})$

lookup(x): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;
falls Eintrag (x, r) gefunden: **return** r ;
 // **erfolgreiche Suche**
sonst: **return** „nicht vorhanden“
 // **erfolglose Suche**

Kosten/Zeit: $O(1 + \text{Zahl der Schlüsselvergleiche „}x = y\text{“})$
... falls die Auswertung von $h(x)$
und ein Schlüsselvergleich Zeit $O(1)$ kostet.

lookup(x): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;
falls Eintrag (x, r) gefunden: **return** r ;
 // **erfolgreiche Suche**
sonst: **return** „nicht vorhanden“
 // **erfolglose Suche**

Kosten/Zeit: $O(1 + \text{Zahl der Schlüsselvergleiche „}x = y\text{“})$

... falls die Auswertung von $h(x)$
und ein Schlüsselvergleich Zeit $O(1)$ kostet.

Erfolglose Suche: Exakt $|B_j|$ Schlüsselvergleiche.

lookup(x): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;
falls Eintrag (x, r) gefunden: **return** r ;
 // **erfolgreiche Suche**
sonst: **return** „nicht vorhanden“
 // **erfolglose Suche**

Kosten/Zeit: $O(1 + \text{Zahl der Schlüsselvergleiche „}x = y\text{“})$

... falls die Auswertung von $h(x)$
und ein Schlüsselvergleich Zeit $O(1)$ kostet.

Erfolglose Suche: Exakt $|B_j|$ Schlüsselvergleiche.

Erfolgreiche Suche: Höchstens $|B_j|$ Schlüsselvergleiche.

insert(x, r): Berechne $j = h(x)$;

insert(x, r): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;

insert(x, r): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;
falls gefunden: // **erfolgreiche Suche**
ersetze Daten bei x durch r ;

insert(x, r): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;
falls gefunden: // **erfolgreiche Suche**
ersetze Daten bei x durch r ;
sonst: // **erfolglose Suche, Normalfall!**
füge (x, r) in Liste $T[j]$ ein.

insert(x, r): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;
falls gefunden: // **erfolgreiche Suche**
ersetze Daten bei x durch r ;
sonst: // **erfolglose Suche, Normalfall!**
füge (x, r) in Liste $T[j]$ ein.

delete(x): Berechne $j = h(x)$;

insert(x, r): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;
falls gefunden: // **erfolgreiche Suche**
ersetze Daten bei x durch r ;
sonst: // **erfolglose Suche, Normalfall!**
füge (x, r) in Liste $T[j]$ ein.

delete(x): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;

insert(x, r): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;
falls gefunden: // **erfolgreiche Suche**
ersetze Daten bei x durch r ;
sonst: // **erfolglose Suche, Normalfall!**
füge (x, r) in Liste $T[j]$ ein.

delete(x): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;
falls gefunden:
// **erfolgreiche Suche, Normalfall!**
lösche Eintrag $(x, f(x))$ aus L_j ;

insert(x, r): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;
falls gefunden: // **erfolgreiche Suche**
ersetze Daten bei x durch r ;
sonst: // **erfolglose Suche, Normalfall!**
füge (x, r) in Liste $T[j]$ ein.

delete(x): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;
falls gefunden:
// **erfolgreiche Suche, Normalfall!**
lösche Eintrag $(x, f(x))$ aus L_j ;
sonst: tue nichts. // **erfolglose Suche**

insert(x, r): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;
falls gefunden: // **erfolgreiche Suche**
ersetze Daten bei x durch r ;
sonst: // **erfolglose Suche, Normalfall!**
füge (x, r) in Liste $T[j]$ ein.

delete(x): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;
falls gefunden:
// **erfolgreiche Suche, Normalfall!**
lösche Eintrag $(x, f(x))$ aus L_j ;
sonst: tue nichts. // **erfolglose Suche**

Kosten jeweils:

insert(x, r): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;
falls gefunden: // **erfolgreiche Suche**
ersetze Daten bei x durch r ;
sonst: // **erfolglose Suche, Normalfall!**
füge (x, r) in Liste $T[j]$ ein.

delete(x): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei $T[j]$;
falls gefunden:
// **erfolgreiche Suche, Normalfall!**
lösche Eintrag $(x, f(x))$ aus L_j ;
sonst: tue nichts. // **erfolglose Suche**

Kosten jeweils: wie bei *lookup*.

Ziel:

Analysiere „**erwartete Kosten**“ für *lookup*, *insert*, *delete*.

Ziel:

Analysiere „**erwartete Kosten**“ für *lookup*, *insert*, *delete*.

Hierfür genügt:

Analysiere „**erwartete Anzahl der Schlüsselvergleiche**“.

Ziel:

Analysiere „**erwartete Kosten**“ für *lookup*, *insert*, *delete*.

Hierfür genügt:

Analysiere „**erwartete Anzahl der Schlüsselvergleiche**“.

Vorausgesetzt: Irgendeine Quelle für „**Zufall**“. (Dazu später mehr.)

Ziel:

Analysiere „**erwartete Kosten**“ für *lookup*, *insert*, *delete*.

Hierfür genügt:

Analysiere „**erwartete Anzahl der Schlüsselvergleiche**“.

Vorausgesetzt: Irgendeine Quelle für „**Zufall**“. (Dazu später mehr.)

Pr(A): Wahrscheinlichkeit für **Ereignis** A .

Ziel:

Analysiere „**erwartete Kosten**“ für *lookup*, *insert*, *delete*.

Hierfür genügt:

Analysiere „**erwartete Anzahl der Schlüsselvergleiche**“.

Vorausgesetzt: Irgendeine Quelle für „**Zufall**“. (Dazu später mehr.)

Pr(A): Wahrscheinlichkeit für **Ereignis** A .

E(X): Erwartungswert der **Zufallsvariablen** X .

Ziel:

Analysiere „**erwartete Kosten**“ für *lookup*, *insert*, *delete*.

Hierfür genügt:

Analysiere „**erwartete Anzahl der Schlüsselvergleiche**“.

Vorausgesetzt: Irgendeine Quelle für „**Zufall**“. (Dazu später mehr.)

Pr(A): Wahrscheinlichkeit für **Ereignis** A .

E(X): Erwartungswert der **Zufallsvariablen** X .

Abgeschwächte „**Uniformitätsannahme (UF₂)**“:

Für je 2 Schlüssel $x \neq z$ in U gilt:

$$\mathbf{Pr}(h(x) = h(z)) \leq \frac{1}{m}$$

0.3 Lineares Sondieren

oder „**offene Adressierung**“: Kollisionsbehandlungs-Methode.

0.3 Lineares Sondieren

oder „offene Adressierung“: Kollisionsbehandlungs-Methode.

Idee: Für jeden Schlüssel $x \in U$ gibt es eine

Sondierungsfolge $h(x, 0), h(x, 1), h(x, 2), \dots$ in $[m]$.

0.3 Lineares Sondieren

oder „**offene Adressierung**“: Kollisionsbehandlungs-Methode.

Idee: Für jeden Schlüssel $x \in U$ gibt es eine

Sondierungsfolge $h(x, 0), h(x, 1), h(x, 2), \dots$ in $[m]$.

Beim **Einfügen** von x und **Suchen** nach x werden die Zellen von T in dieser Reihenfolge untersucht, bis eine leere Zelle oder der Eintrag x gefunden wird.

0.3 Lineares Sondieren

oder „**offene Adressierung**“: Kollisionsbehandlungs-Methode.

Idee: Für jeden Schlüssel $x \in U$ gibt es eine

Sondierungsfolge $h(x, 0), h(x, 1), h(x, 2), \dots$ in $[m]$.

Beim **Einfügen** von x und **Suchen** nach x werden die Zellen von T in dieser Reihenfolge untersucht, bis eine leere Zelle oder der Eintrag x gefunden wird. – Günstig:

0.3 Lineares Sondieren

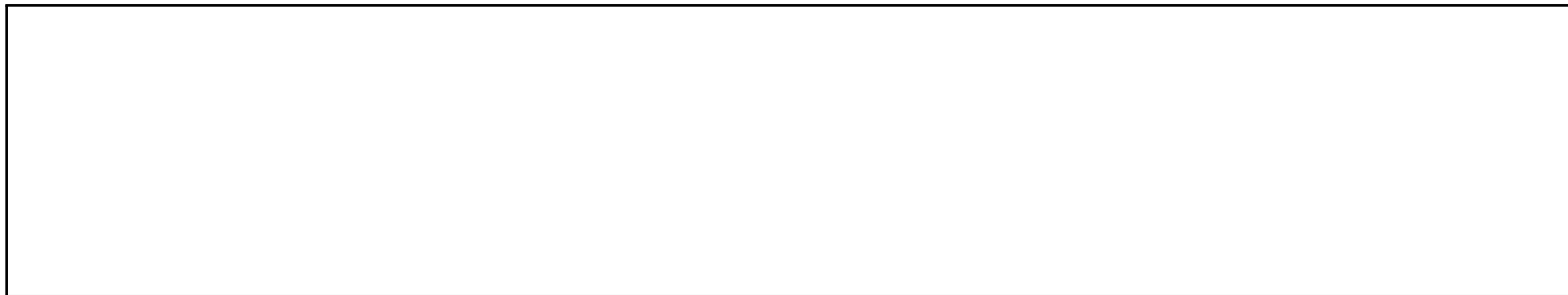
oder „**offene Adressierung**“: Kollisionsbehandlungs-Methode.

Idee: Für jeden Schlüssel $x \in U$ gibt es eine

Sondierungsfolge $h(x, 0), h(x, 1), h(x, 2), \dots$ in $[m]$.

Beim **Einfügen** von x und **Suchen** nach x werden die Zellen von T in dieser Reihenfolge untersucht, bis eine leere Zelle oder der Eintrag x gefunden wird. – Günstig:

(*)



0.3 Lineares Sondieren

oder „**offene Adressierung**“: Kollisionsbehandlungs-Methode.

Idee: Für jeden Schlüssel $x \in U$ gibt es eine

Sondierungsfolge $h(x, 0), h(x, 1), h(x, 2), \dots$ in $[m]$.

Beim **Einfügen** von x und **Suchen** nach x werden die Zellen von T in dieser Reihenfolge untersucht, bis eine leere Zelle oder der Eintrag x gefunden wird. – Günstig:

(*) $h(x, 0), h(x, 1), \dots, h(x, m - 1)$ erreicht jede Zelle,

0.3 Lineares Sondieren

oder „**offene Adressierung**“: Kollisionsbehandlungs-Methode.

Idee: Für jeden Schlüssel $x \in U$ gibt es eine

Sondierungsfolge $h(x, 0), h(x, 1), h(x, 2), \dots$ in $[m]$.

Beim **Einfügen** von x und **Suchen** nach x werden die Zellen von T in dieser Reihenfolge untersucht, bis eine leere Zelle oder der Eintrag x gefunden wird. – Günstig:

$h(x, 0), h(x, 1), \dots, h(x, m - 1)$ erreicht jede Zelle,
(* d. h.: $(h(x, 0), h(x, 1), h(x, 2), \dots, h(x, m - 1))$
ist eine **Permutation** von $[m]$.

0.3 Lineares Sondieren

oder „**offene Adressierung**“: Kollisionsbehandlungs-Methode.

Idee: Für jeden Schlüssel $x \in U$ gibt es eine

Sondierungsfolge $h(x, 0), h(x, 1), h(x, 2), \dots$ in $[m]$.

Beim **Einfügen** von x und **Suchen** nach x werden die Zellen von T in dieser Reihenfolge untersucht, bis eine leere Zelle oder der Eintrag x gefunden wird. – Günstig:

$h(x, 0), h(x, 1), \dots, h(x, m - 1)$ erreicht jede Zelle,
(* d. h.: $(h(x, 0), h(x, 1), h(x, 2), \dots, h(x, m - 1))$
ist eine **Permutation** von $[m]$).

Wenn (*) gilt und mindestens eine Zelle in T leer ist, dann endet jede erfolglose Suche in einer leeren Zelle.

Lineares Sondieren

Lineares Sondieren

$$h(x, k) = (h(x) + k) \bmod m, k = 0, 1, 2, \dots$$

Lineares Sondieren

$$h(x, k) = (h(x) + k) \bmod m, \quad k = 0, 1, 2, \dots$$

Offensichtlich ist (*) erfüllt.

Lineares Sondieren

$$h(x, k) = (h(x) + k) \bmod m, k = 0, 1, 2, \dots$$

Offensichtlich ist (*) erfüllt.

Nächste Tabelle: „Wunschplätze“ für die Schlüssel.

Lineares Sondieren

$$h(x, k) = (h(x) + k) \bmod m, k = 0, 1, 2, \dots$$

Offensichtlich ist (*) erfüllt.

Nächste Tabelle: „Wunschplätze“ für die Schlüssel.

Übernächste Tabelle: Einfügen gemäß linearem Sondieren.

$$h(c_1 \dots c_r) = (\text{num}(c_3) + 11) \bmod 13$$

j	x
0	September ⁽⁰⁾
1	Maerz ⁽¹⁾ April ⁽¹⁾
2	
3	
4	August ⁽⁴⁾ Oktober ⁽⁴⁾
5	
6	Mai ⁽⁶⁾ November ⁽⁶⁾
7	
8	
9	Juli ⁽⁹⁾
10	Dezember ⁽¹⁰⁾
11	Januar ⁽¹¹⁾ Juni ⁽¹¹⁾
12	Februar ⁽¹²⁾

Tabelle T[0..12]:

Tabelle T[0..12]:

Platz	Eintrag
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	

Tabelle T[0..12]:

Platz	Eintrag
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	Januar ⁽¹¹⁾
12	

Tabelle T[0..12]:

Platz	Eintrag
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	Januar ⁽¹¹⁾
12	Februar ⁽¹²⁾

Tabelle T[0..12]:

Platz	Eintrag
0	
1	Maerz ⁽¹⁾
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	Januar ⁽¹¹⁾
12	Februar ⁽¹²⁾

Tabelle T[0..12]:

Platz	Eintrag
0	
1	Maerz ⁽¹⁾ April ⁽¹⁾
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	Januar ⁽¹¹⁾
12	Februar ⁽¹²⁾

Tabelle T[0..12]:

Platz	Eintrag
0	
1	Maerz ⁽¹⁾
2	April ⁽¹⁾
3	
4	
5	
6	
7	
8	
9	
10	
11	Januar ⁽¹¹⁾
12	Februar ⁽¹²⁾

Tabelle T[0..12]:

Platz	Eintrag
0	
1	Maerz ⁽¹⁾
2	April ⁽¹⁾
3	
4	
5	
6	Mai ⁽⁶⁾
7	
8	
9	
10	
11	Januar ⁽¹¹⁾
12	Februar ⁽¹²⁾

Tabelle T[0..12]:

Platz	Eintrag
0	
1	Maerz ⁽¹⁾
2	April ⁽¹⁾
3	
4	
5	
6	Mai ⁽⁶⁾
7	
8	
9	
10	
11	Januar ⁽¹¹⁾ Juni ⁽¹¹⁾
12	Februar ⁽¹²⁾

Tabelle T[0..12]:

Platz	Eintrag
0	
1	Maerz ⁽¹⁾
2	April ⁽¹⁾
3	
4	
5	
6	Mai ⁽⁶⁾
7	
8	
9	
10	
11	Januar ⁽¹¹⁾
12	Februar ⁽¹²⁾ Juni ⁽¹¹⁾

Tabelle T[0..12]:

Platz	Eintrag
0	Juni ⁽¹¹⁾
1	Maerz ⁽¹⁾
2	April ⁽¹⁾
3	
4	
5	
6	Mai ⁽⁶⁾
7	
8	
9	
10	
11	Januar ⁽¹¹⁾
12	Februar ⁽¹²⁾

Tabelle T[0..12]:

Platz	Eintrag
0	Juni ⁽¹¹⁾
1	Maerz ⁽¹⁾
2	April ⁽¹⁾
3	
4	
5	
6	Mai ⁽⁶⁾
7	
8	
9	Juli ⁽⁹⁾
10	
11	Januar ⁽¹¹⁾
12	Februar ⁽¹²⁾

Tabelle T[0..12]:

Platz	Eintrag
0	Juni ⁽¹¹⁾
1	Maerz ⁽¹⁾
2	April ⁽¹⁾
3	
4	August ⁽⁴⁾
5	
6	Mai ⁽⁶⁾
7	
8	
9	Juli ⁽⁹⁾
10	
11	Januar ⁽¹¹⁾
12	Februar ⁽¹²⁾

Tabelle T[0..12]:

Platz	Eintrag
0	Juni ⁽¹¹⁾
1	Maerz ⁽¹⁾
2	April ⁽¹⁾
3	September ⁽⁰⁾
4	August ⁽⁴⁾
5	
6	Mai ⁽⁶⁾
7	
8	
9	Juli ⁽⁹⁾
10	
11	Januar ⁽¹¹⁾
12	Februar ⁽¹²⁾

Tabelle T[0..12]:

Platz	Eintrag
0	Juni ⁽¹¹⁾
1	Maerz ⁽¹⁾
2	April ⁽¹⁾
3	September ⁽⁰⁾
4	August ⁽⁴⁾
5	Oktober ⁽⁴⁾
6	Mai ⁽⁶⁾
7	
8	
9	Juli ⁽⁹⁾
10	
11	Januar ⁽¹¹⁾
12	Februar ⁽¹²⁾

Tabelle T[0..12]:

Platz	Eintrag
0	Juni ⁽¹¹⁾
1	Maerz ⁽¹⁾
2	April ⁽¹⁾
3	September ⁽⁰⁾
4	August ⁽⁴⁾
5	Oktober ⁽⁴⁾
6	Mai ⁽⁶⁾
7	November ⁽⁶⁾
8	
9	Juli ⁽⁹⁾
10	
11	Januar ⁽¹¹⁾
12	Februar ⁽¹²⁾

Tabelle T[0..12]:

Platz	Eintrag
0	Juni ⁽¹¹⁾
1	Maerz ⁽¹⁾
2	April ⁽¹⁾
3	September ⁽⁰⁾
4	August ⁽⁴⁾
5	Oktober ⁽⁴⁾
6	Mai ⁽⁶⁾
7	November ⁽⁶⁾
8	
9	Juli ⁽⁹⁾
10	Dezember ⁽¹⁰⁾
11	Januar ⁽¹¹⁾
12	Februar ⁽¹²⁾

empty(m): Lege leeres Array $T[0..m-1]$ an.

empty(m): Lege leeres Array $T[0..m-1]$ an.
Arrayeintrag: (Schlüssel, Wert (in R)) oder *leer*.

empty(m): Lege leeres Array $T[0 \dots m - 1]$ an.
Arrayeintrag: (Schlüssel, Wert (in R)) oder *leer*.
Initialisiere alle Plätze $T[j]$ mit *leer*.

empty(m): Lege leeres Array $T[0 \dots m - 1]$ an.
Arrayeintrag: (Schlüssel, Wert (in R)) oder *leer*.
Initialisiere alle Plätze $T[j]$ mit *leer*.
Initialisiere *load* mit 0, *maxload* mit Wert $< m$.

empty(m): Lege leeres Array $T[0..m-1]$ an.

Arrayeintrag: (Schlüssel, Wert (in R)) oder *leer*.

Initialisiere alle Plätze $T[j]$ mit *leer*.

Initialisiere *load* mit 0, *maxload* mit Wert $< m$.

insert(x, r): $j \leftarrow h(x)$;

Finde erstes l in der Folge $j, (j+1) \bmod m, (j+2) \bmod m, \dots$
mit $T[l].\text{key} = x$ **oder** $T[l] = \textit{leer}$.

empty(m): Lege leeres Array $T[0..m-1]$ an.

Arrayeintrag: (Schlüssel, Wert (in R)) oder *leer*.

Initialisiere alle Plätze $T[j]$ mit *leer*.

Initialisiere *load* mit 0, *maxload* mit Wert $< m$.

insert(x, r): $j \leftarrow h(x)$;

Finde erstes l in der Folge $j, (j+1) \bmod m, (j+2) \bmod m, \dots$
mit $T[l].\text{key} = x$ **oder** $T[l] = \text{leer}$.

Im Fall $T[l].\text{key} = x$: // **erfolgreiche Suche**

$T[l].\text{data} \leftarrow r$;

empty(m): Lege leeres Array $T[0..m-1]$ an.

Arrayeintrag: (Schlüssel, Wert (in R)) oder *leer*.

Initialisiere alle Plätze $T[j]$ mit *leer*.

Initialisiere *load* mit 0, *maxload* mit Wert $< m$.

insert(x, r): $j \leftarrow h(x)$;

Finde erstes l in der Folge $j, (j+1) \bmod m, (j+2) \bmod m, \dots$

mit $T[l].\text{key} = x$ **oder** $T[l] = \textit{leer}$.

Im Fall $T[l].\text{key} = x$: // **erfolgreiche Suche**

$T[l].\text{data} \leftarrow r$;

Im Fall $T[l] = \textit{leer}$: // **erfolglose Suche**

empty(m): Lege leeres Array $T[0..m-1]$ an.

Arrayeintrag: (Schlüssel, Wert (in R)) oder *leer*.

Initialisiere alle Plätze $T[j]$ mit *leer*.

Initialisiere *load* mit 0, *maxload* mit Wert $< m$.

insert(x, r): $j \leftarrow h(x)$;

Finde erstes l in der Folge $j, (j+1) \bmod m, (j+2) \bmod m, \dots$

mit $T[l].\text{key} = x$ **oder** $T[l] = \text{leer}$.

Im Fall $T[l].\text{key} = x$: // **erfolgreiche Suche**

$T[l].\text{data} \leftarrow r$;

Im Fall $T[l] = \text{leer}$: // **erfolglose Suche**

Falls $\text{load} = \text{maxload}$: „Überlaufbehandlung“

(z. B. Verdopplung von T , Neustart von *insert*(x, r));

empty(m): Lege leeres Array $T[0..m-1]$ an.

Arrayeintrag: (Schlüssel, Wert (in R)) oder *leer*.

Initialisiere alle Plätze $T[j]$ mit *leer*.

Initialisiere *load* mit 0, *maxload* mit Wert $< m$.

insert(x, r): $j \leftarrow h(x)$;

Finde erstes l in der Folge $j, (j+1) \bmod m, (j+2) \bmod m, \dots$
mit $T[l].\text{key} = x$ **oder** $T[l] = \text{leer}$.

Im Fall $T[l].\text{key} = x$: // **erfolgreiche Suche**

$T[l].\text{data} \leftarrow r$;

Im Fall $T[l] = \text{leer}$: // **erfolglose Suche**

Falls $\text{load} = \text{maxload}$: „Überlaufbehandlung“

(z. B. Verdopplung von T , Neustart von *insert*(x, r));

Sonst: $T[l] \leftarrow (x, r)$; $\text{load}++$.

lookup(x): $j \leftarrow h(x)$;

Finde erstes l in der Folge $j, (j + 1) \bmod m, (j + 2) \bmod m, \dots$
mit $T[l].\text{key} = x$ **oder** $T[l] = \text{leer}$.

lookup(x): $j \leftarrow h(x)$;

Finde erstes l in der Folge $j, (j + 1) \bmod m, (j + 2) \bmod m, \dots$

mit $T[l].\text{key} = x$ **oder** $T[l] = \text{leer}$.

Im Fall $T[l].\text{key} = x$: **return** $T[l].\text{data}$.

// **erfolgreiche Suche**

lookup(x): $j \leftarrow h(x)$;

Finde erstes l in der Folge $j, (j + 1) \bmod m, (j + 2) \bmod m, \dots$
mit $T[l].\text{key} = x$ **oder** $T[l] = \text{leer}$.

Im Fall $T[l].\text{key} = x$: **return** $T[l].\text{data}$.

// **erfolgreiche Suche**

Im Fall $T[l] = \text{leer}$: **return** „nicht vorhanden“.

// **erfolglose Suche**

lookup(x): $j \leftarrow h(x)$;

Finde erstes l in der Folge $j, (j + 1) \bmod m, (j + 2) \bmod m, \dots$
mit $T[l].key = x$ **oder** $T[l] = leer$.

Im Fall $T[l].key = x$: **return** $T[l].data$.

// **erfolgreiche Suche**

Im Fall $T[l] = leer$: **return** „nicht vorhanden“.

// **erfolglose Suche**

Korrektheit der Suche: Bei *lookup*(x) werden genau dieselben Zellen von T untersucht wie beim Einfügen von x .

lookup(x): $j \leftarrow h(x)$;

Finde erstes l in der Folge $j, (j + 1) \bmod m, (j + 2) \bmod m, \dots$
mit $T[l].\text{key} = x$ **oder** $T[l] = \text{leer}$.

Im Fall $T[l].\text{key} = x$: **return** $T[l].\text{data}$.

// **erfolgreiche Suche**

Im Fall $T[l] = \text{leer}$: **return** „nicht vorhanden“.

// **erfolglose Suche**

Korrektheit der Suche: Bei *lookup*(x) werden genau dieselben Zellen von T untersucht wie beim Einfügen von x .

Da kein Element bewegt/gelöscht wird, wird auf diese Weise die Zelle, in der x sitzt, wieder erreicht, falls vorhanden

lookup(x): $j \leftarrow h(x)$;

Finde erstes l in der Folge $j, (j + 1) \bmod m, (j + 2) \bmod m, \dots$
mit $T[l].\text{key} = x$ **oder** $T[l] = \text{leer}$.

Im Fall $T[l].\text{key} = x$: **return** $T[l].\text{data}$.

// **erfolgreiche Suche**

Im Fall $T[l] = \text{leer}$: **return** „nicht vorhanden“.

// **erfolglose Suche**

Korrektheit der Suche: Bei *lookup*(x) werden genau dieselben Zellen von T untersucht wie beim Einfügen von x .

Da kein Element bewegt/gelöscht wird, wird auf diese Weise die Zelle, in der x sitzt, wieder erreicht, falls vorhanden

oder eine leere Zelle –

lookup(x): $j \leftarrow h(x)$;

Finde erstes l in der Folge $j, (j + 1) \bmod m, (j + 2) \bmod m, \dots$
mit $T[l].\text{key} = x$ **oder** $T[l] = \text{leer}$.

Im Fall $T[l].\text{key} = x$: **return** $T[l].\text{data}$.

// **erfolgreiche Suche**

Im Fall $T[l] = \text{leer}$: **return** „nicht vorhanden“.

// **erfolglose Suche**

Korrektheit der Suche: Bei *lookup*(x) werden genau dieselben Zellen von T untersucht wie beim Einfügen von x .

Da kein Element bewegt/gelöscht wird, wird auf diese Weise die Zelle, in der x sitzt, wieder erreicht, falls vorhanden

oder eine leere Zelle – wir müssen nur sicherstellen, dass **immer eine leere Zelle** vorhanden ist, also $\text{load} \leq m - 1$ ist.

Erinnerung:

Erinnerung:

„Große“ Uniformitätsannahme (UF*):

Die Werte $h(x_1), \dots, h(x_n)$ sind **rein zufällig**.

Proposition 0.3.1 Ohne Beweis.

Proposition 0.3.1 Ohne Beweis.

Bei linearem Sondieren in einer Tabelle der Größe m mit Schlüsselmenge $S = \{x_1, \dots, x_n\}$ gilt, mit $\alpha = \frac{n}{m}$:

Proposition 0.3.1 Ohne Beweis.

Bei linearem Sondieren in einer Tabelle der Größe m mit Schlüsselmenge $S = \{x_1, \dots, x_n\}$ gilt, mit $\alpha = \frac{n}{m}$:

Die **erwartete** (nach (UF*)) Anzahl von Schlüsselvergleichen (besuchte Arrayzellen) bei **erfolgloser Suche** nach $y \notin S$ ist

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

Proposition 0.3.1 Ohne Beweis.

Bei linearem Sondieren in einer Tabelle der Größe m mit Schlüsselmenge $S = \{x_1, \dots, x_n\}$ gilt, mit $\alpha = \frac{n}{m}$:

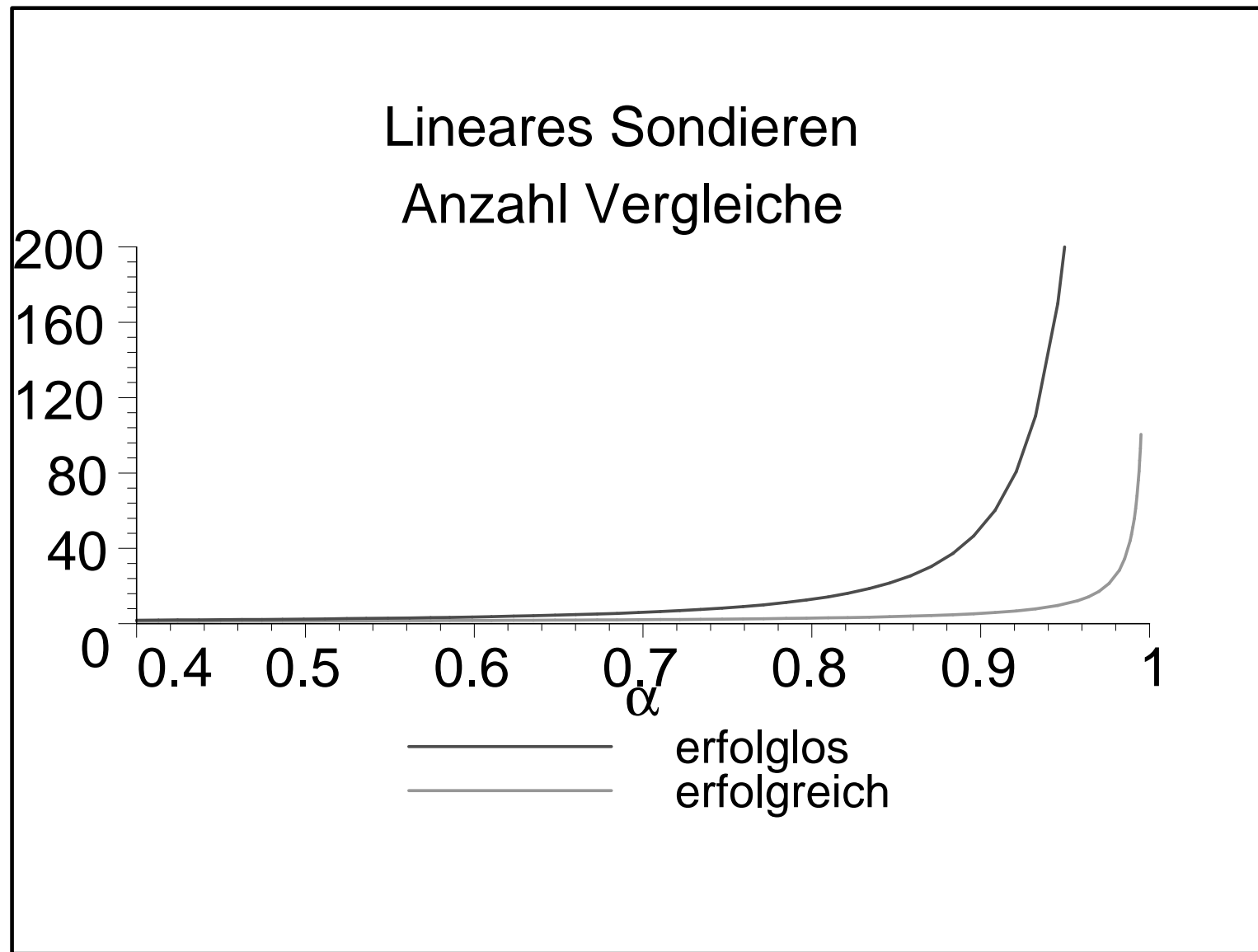
Die **erwartete** (nach (UF*)) Anzahl von Schlüsselvergleichen (besuchte Arrayzellen) bei **erfolgloser Suche** nach $y \notin S$ ist

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

Die **erwartete** (nach (UF*)) **mittlere** (über x_1, \dots, x_n) Anzahl von Schlüsselvergleichen bei **erfolgreicher Suche** nach $y \in S$ ist

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

An der Tafel: Für $\alpha = \frac{1}{2}$ erwartete Einfügezeit: $O(1)$..



Erwarteter Aufwand ist also $O(1)$, falls $\alpha \leq \alpha_0$ für eine Konstante $\alpha_0 < 1$.

Erwarteter Aufwand ist also $O(1)$, falls $\alpha \leq \alpha_0$ für eine Konstante $\alpha_0 < 1$.

Erwartete Anzahl von Schlüsselvergleichen bei linearem Sondieren:

Erwarteter Aufwand ist also $O(1)$, falls $\alpha \leq \alpha_0$ für eine Konstante $\alpha_0 < 1$.

Erwartete Anzahl von Schlüsselvergleichen bei linearem Sondieren:

	erfolglose Suche	erfolgreiche Suche
α	$\frac{1}{2} \cdot \left(1 + \frac{1}{(1-\alpha)^2}\right)$	$\frac{1}{2} \cdot \left(1 + \frac{1}{1-\alpha}\right)$

Erwarteter Aufwand ist also $O(1)$, falls $\alpha \leq \alpha_0$ für eine Konstante $\alpha_0 < 1$.

Erwartete Anzahl von Schlüsselvergleichen bei linearem Sondieren:

	erfolglose Suche	erfolgreiche Suche
α	$\frac{1}{2} \cdot \left(1 + \frac{1}{(1-\alpha)^2}\right)$	$\frac{1}{2} \cdot \left(1 + \frac{1}{1-\alpha}\right)$
0,5	2,5	1,5

Erwarteter Aufwand ist also $O(1)$, falls $\alpha \leq \alpha_0$ für eine Konstante $\alpha_0 < 1$.

Erwartete Anzahl von Schlüsselvergleichen bei linearem Sondieren:

	erfolglose Suche	erfolgreiche Suche
α	$\frac{1}{2} \cdot \left(1 + \frac{1}{(1-\alpha)^2}\right)$	$\frac{1}{2} \cdot \left(1 + \frac{1}{1-\alpha}\right)$
0,5	2,5	1,5
0,6	3,625	1,75

Erwarteter Aufwand ist also $O(1)$, falls $\alpha \leq \alpha_0$ für eine Konstante $\alpha_0 < 1$.

Erwartete Anzahl von Schlüsselvergleichen bei linearem Sondieren:

	erfolglose Suche	erfolgreiche Suche
α	$\frac{1}{2} \cdot \left(1 + \frac{1}{(1-\alpha)^2}\right)$	$\frac{1}{2} \cdot \left(1 + \frac{1}{1-\alpha}\right)$
0,5	2,5	1,5
0,6	3,625	1,75
0,7	6,06	2,16

Erwarteter Aufwand ist also $O(1)$, falls $\alpha \leq \alpha_0$ für eine Konstante $\alpha_0 < 1$.

Erwartete Anzahl von Schlüsselvergleichen bei linearem Sondieren:

	erfolglose Suche	erfolgreiche Suche
α	$\frac{1}{2} \cdot \left(1 + \frac{1}{(1-\alpha)^2}\right)$	$\frac{1}{2} \cdot \left(1 + \frac{1}{1-\alpha}\right)$
0,5	2,5	1,5
0,6	3,625	1,75
0,7	6,06	2,16
0,75	8,5	2,5

Erwarteter Aufwand ist also $O(1)$, falls $\alpha \leq \alpha_0$ für eine Konstante $\alpha_0 < 1$.

Erwartete Anzahl von Schlüsselvergleichen bei linearem Sondieren:

	erfolglose Suche	erfolgreiche Suche
α	$\frac{1}{2} \cdot \left(1 + \frac{1}{(1-\alpha)^2}\right)$	$\frac{1}{2} \cdot \left(1 + \frac{1}{1-\alpha}\right)$
0,5	2,5	1,5
0,6	3,625	1,75
0,7	6,06	2,16
0,75	8,5	2,5
0,8	13	3

Erwarteter Aufwand ist also $O(1)$, falls $\alpha \leq \alpha_0$ für eine Konstante $\alpha_0 < 1$.

Erwartete Anzahl von Schlüsselvergleichen bei linearem Sondieren:

	erfolglose Suche	erfolgreiche Suche
α	$\frac{1}{2} \cdot \left(1 + \frac{1}{(1-\alpha)^2}\right)$	$\frac{1}{2} \cdot \left(1 + \frac{1}{1-\alpha}\right)$
0,5	2,5	1,5
0,6	3,625	1,75
0,7	6,06	2,16
0,75	8,5	2,5
0,8	13	3
0,9	50,5	5,5
0,95	200,5	20

Ergebnisse für lineares Sondieren:

Ergebnisse für lineares Sondieren:

Eine einzelne erfolglose Suche wird z. B. bei zu 90% gefüllter Tabelle schon recht teuer.

Ergebnisse für lineares Sondieren:

Eine einzelne erfolglose Suche wird z. B. bei zu 90% gefüllter Tabelle schon recht teuer.

Bei vielen erfolglosen Suchen in einer nicht veränderlichen Tabelle (anwendungsabhängig!) sind diese Kosten nicht tolerierbar.

Ergebnisse für lineares Sondieren:

Eine einzelne erfolglose Suche wird z. B. bei zu 90% gefüllter Tabelle schon recht teuer.

Bei vielen erfolglosen Suchen in einer nicht veränderlichen Tabelle (anwendungsabhängig!) sind diese Kosten nicht tolerierbar.

$\alpha \leq 0,6$ oder $\alpha \leq 0,7$ liefert jedoch akzeptable Kosten.

Ergebnisse für lineares Sondieren:

Eine einzelne erfolglose Suche wird z. B. bei zu 90% gefüllter Tabelle schon recht teuer.

Bei vielen erfolglosen Suchen in einer nicht veränderlichen Tabelle (anwendungsabhängig!) sind diese Kosten nicht tolerierbar.

$\alpha \leq 0,6$ oder $\alpha \leq 0,7$ liefert jedoch akzeptable Kosten.

Die (über $y \in S$) **gemittelte** Einfüge-/Suchzeit ist dagegen sogar bei 90% gefüllter Tabelle noch erträglich.

Ergebnisse für lineares Sondieren:

Eine einzelne erfolglose Suche wird z. B. bei zu 90% gefüllter Tabelle schon recht teuer.

Bei vielen erfolglosen Suchen in einer nicht veränderlichen Tabelle (anwendungsabhängig!) sind diese Kosten nicht tolerierbar.

$\alpha \leq 0,6$ oder $\alpha \leq 0,7$ liefert jedoch akzeptable Kosten.

Die (über $y \in S$) **gemittelte** Einfüge-/Suchzeit ist dagegen sogar bei 90% gefüllter Tabelle noch erträglich.

!!!

Ergebnisse für lineares Sondieren:

Eine einzelne erfolglose Suche wird z. B. bei zu 90% gefüllter Tabelle schon recht teuer.

Bei vielen erfolglosen Suchen in einer nicht veränderlichen Tabelle (anwendungsabhängig!) sind diese Kosten nicht tolerierbar.

$\alpha \leq 0,6$ oder $\alpha \leq 0,7$ liefert jedoch akzeptable Kosten.

Die (über $y \in S$) **gemittelte** Einfüge-/Suchzeit ist dagegen sogar bei 90% gefüllter Tabelle noch erträglich.

Lineares Sondieren passt perfekt zu **Cache-Architekturen**

!!!

Ergebnisse für lineares Sondieren:

Eine einzelne erfolglose Suche wird z. B. bei zu 90% gefüllter Tabelle schon recht teuer.

Bei vielen erfolglosen Suchen in einer nicht veränderlichen Tabelle (anwendungsabhängig!) sind diese Kosten nicht tolerierbar.

$\alpha \leq 0,6$ oder $\alpha \leq 0,7$ liefert jedoch akzeptable Kosten.

Die (über $y \in S$) **gemittelte** Einfüge-/Suchzeit ist dagegen sogar bei 90% gefüllter Tabelle noch erträglich.

!!! Lineares Sondieren passt perfekt zu **Cache-Architekturen** und ist deswegen auch in der Praxis sehr schnell.

Ergebnisse für lineares Sondieren:

Eine einzelne erfolglose Suche wird z. B. bei zu 90% gefüllter Tabelle schon recht teuer.

Bei vielen erfolglosen Suchen in einer nicht veränderlichen Tabelle (anwendungsabhängig!) sind diese Kosten nicht tolerierbar.

$\alpha \leq 0,6$ oder $\alpha \leq 0,7$ liefert jedoch akzeptable Kosten.

Die (über $y \in S$) **gemittelte** Einfüge-/Suchzeit ist dagegen sogar bei 90% gefüllter Tabelle noch erträglich.

Lineares Sondieren passt perfekt zu **Cache-Architekturen**

!!!

und ist deswegen auch in der Praxis sehr schnell.

(Bei einem Cachefehler wird eine ganze „Zeile“ in den Cache geholt, also mehrere aufeinanderfolgende Wörter aus T.)

Verdoppelungsstrategie bei linearem Sondieren:

Verdoppelungsstrategie bei linearem Sondieren:

Verdopple die Tabelle, sobald der Auslastungsfaktor $\alpha = n/m$ einen Wert von α_0 übersteigt – z.B. $\alpha_0 = 0,75$ oder $\alpha_0 = 0,8$.

Verdoppelungsstrategie bei linearem Sondieren:

Verdopple die Tabelle, sobald der Auslastungsfaktor $\alpha = n/m$ einen Wert von α_0 übersteigt – z.B. $\alpha_0 = 0,75$ oder $\alpha_0 = 0,8$.

Empfehlung: Kombiniere lineares Sondieren mit einer guten universellen Hashklasse, z. B. Tabellenhashing.

Verdoppelungsstrategie bei linearem Sondieren:

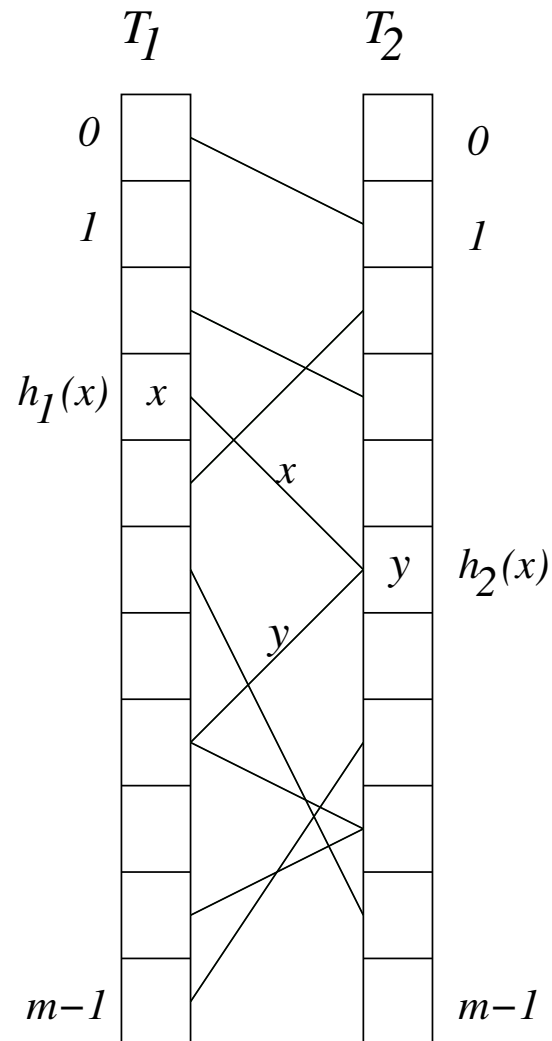
Verdopple die Tabelle, sobald der Auslastungsfaktor $\alpha = n/m$ einen Wert von α_0 übersteigt – z.B. $\alpha_0 = 0,75$ oder $\alpha_0 = 0,8$.

Empfehlung: Kombiniere lineares Sondieren mit einer guten universellen Hashklasse, z. B. Tabellenhashing.

Nicht auf die Zufälligkeit von S vertrauen!

0.4 Cuckoo Hashing [Pagh/Rodler 2001/04]

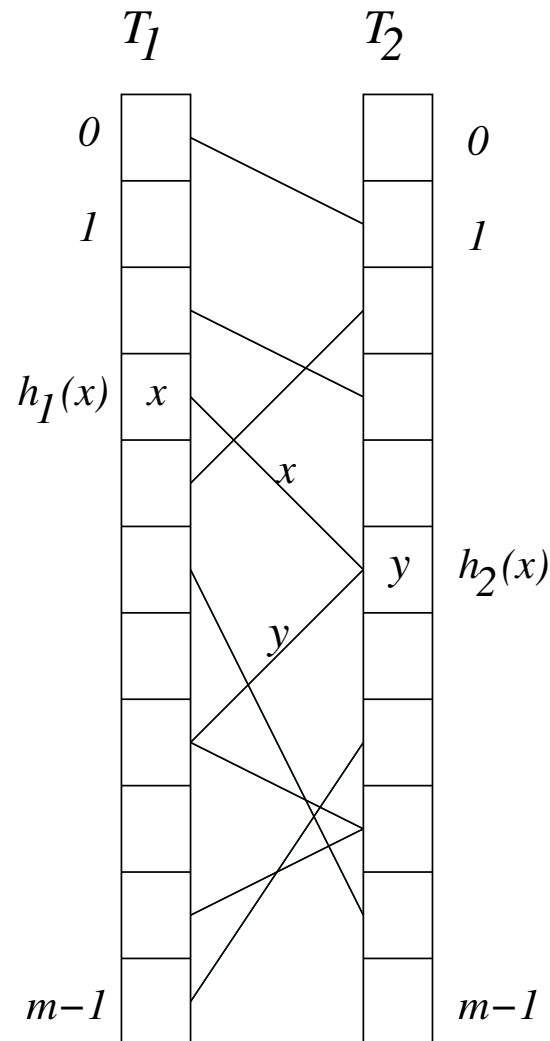
0.4 Cuckoo Hashing [Pagh/Rodler 2001/04]



Implementierung eines dynamischen Wörterbuchs
Zwei Tabellen T_1, T_2 ,
jeweils Größe m

$x \in S$
in $T_1[h_1(x)]$ **oder**
in $T_2[h_2(x)]$.

0.4 Cuckoo Hashing [Pagh/Rodler 2001/04]

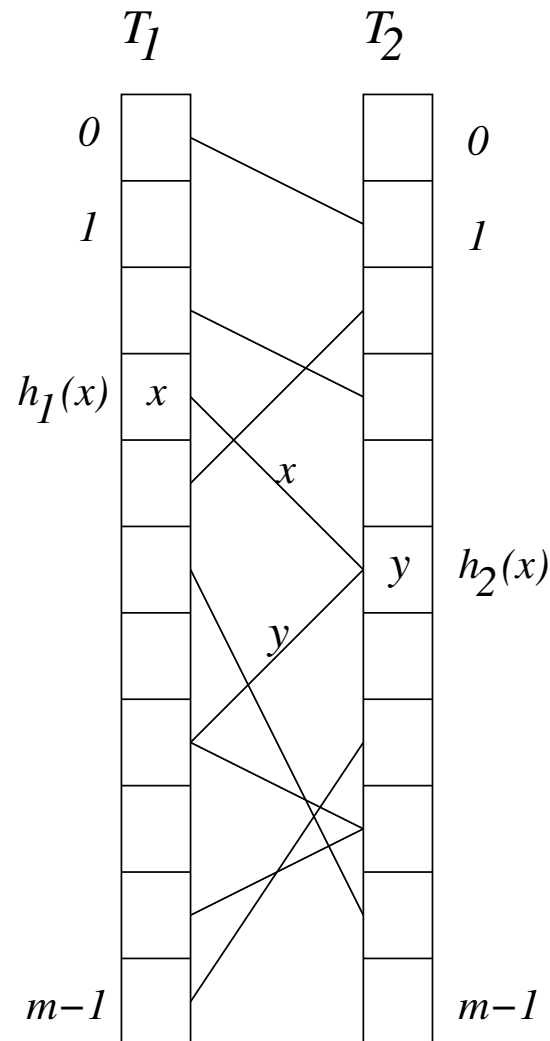


Implementierung eines dynamisches Wörterbuchs
Zwei Tabellen T_1, T_2 ,
jeweils Größe m

$x \in S$
in $T_1[h_1(x)]$ **oder**
in $T_2[h_2(x)]$.

\Rightarrow **Konstante Suchzeit**
garantiert.

0.4 Cuckoo Hashing [Pagh/Rodler 2001/04]



Implementierung eines dynamisches Wörterbuchs
Zwei Tabellen T_1, T_2 ,
jeweils Größe m

$x \in S$
in $T_1[h_1(x)]$ **oder**
in $T_2[h_2(x)]$.

\Rightarrow **Konstante Suchzeit**
garantiert.

Auch $delete(x)$ in $O(1)$ Zeit.

Cuckoo-Hashing

Definition 0.4.1

h_1, h_2 **passen zu** S , falls die Schlüssel aus S gemäß der Regel gespeichert werden können: man kann S in S_1 und S_2 partitionieren, so dass h_1 auf S_1 und h_2 auf S_2 injektiv ist.

Cuckoo-Hashing

Definition 0.4.1

h_1, h_2 **passen zu** S , falls die Schlüssel aus S gemäß der Regel gespeichert werden können: man kann S in S_1 und S_2 partitionieren, so dass h_1 auf S_1 und h_2 auf S_2 injektiv ist.

“Kuckucks-Hashing” wegen interessanter Einfügeprozedur:

Cuckoo-Hashing

Definition 0.4.1

h_1, h_2 **passen zu** S , falls die Schlüssel aus S gemäß der Regel gespeichert werden können: man kann S in S_1 und S_2 partitionieren, so dass h_1 auf S_1 und h_2 auf S_2 injektiv ist.

“Kuckucks-Hashing” wegen interessanter Einfügeprozedur:

Schlüssel x , der eingefügt werden soll, kann Schlüssel y , der in $T_1[h_1(x)]$ oder $T_2[h_2(x)]$ („im Nest“) sitzt,

hinauswerfen (verdrängen).

Cuckoo-Hashing

Definition 0.4.1

h_1, h_2 **passen zu** S , falls die Schlüssel aus S gemäß der Regel gespeichert werden können: man kann S in S_1 und S_2 partitionieren, so dass h_1 auf S_1 und h_2 auf S_2 injektiv ist.

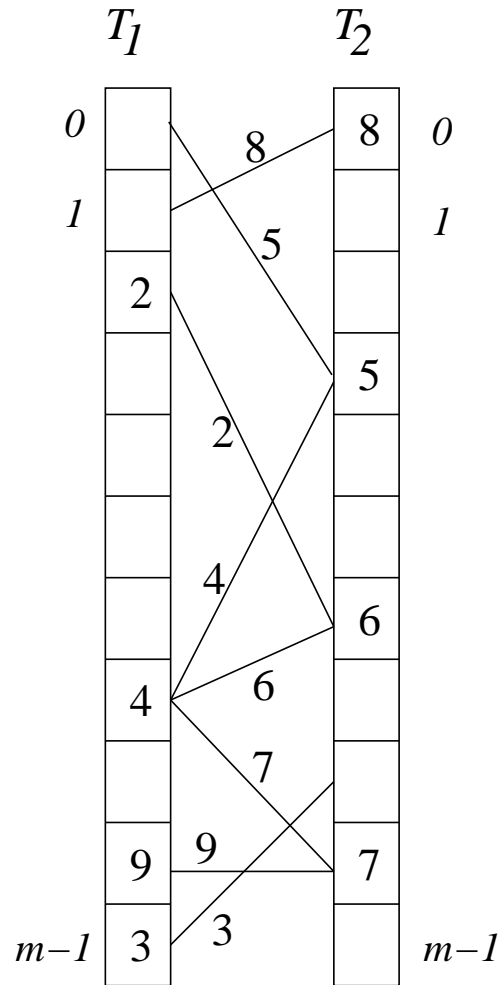
“Kuckucks-Hashing” wegen interessanter Einfügeprozedur:

Schlüssel x , der eingefügt werden soll, kann Schlüssel y , der in $T_1[h_1(x)]$ oder $T_2[h_2(x)]$ („im Nest“) sitzt,

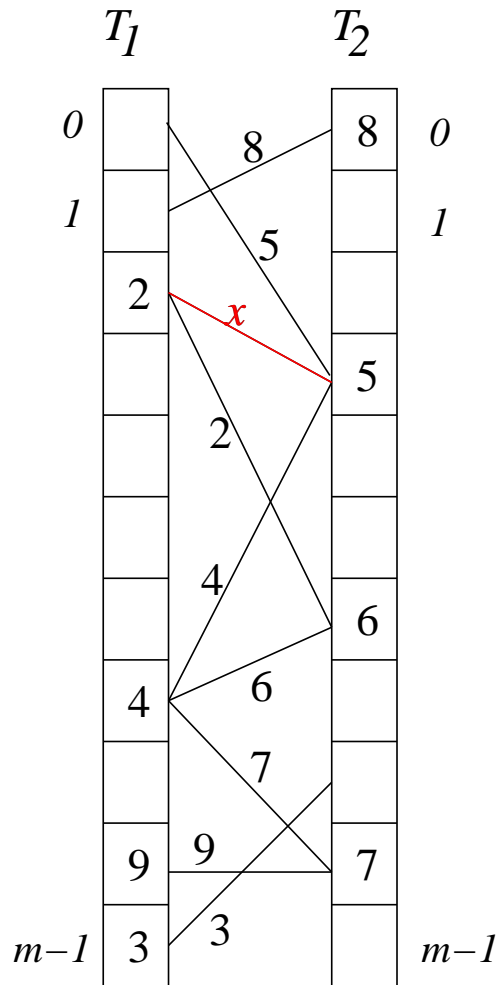
hinauswerfen (verdrängen).

Der verdrängte Schlüssel geht zu seinem alternativen Platz.

Cuckoo-Hashing

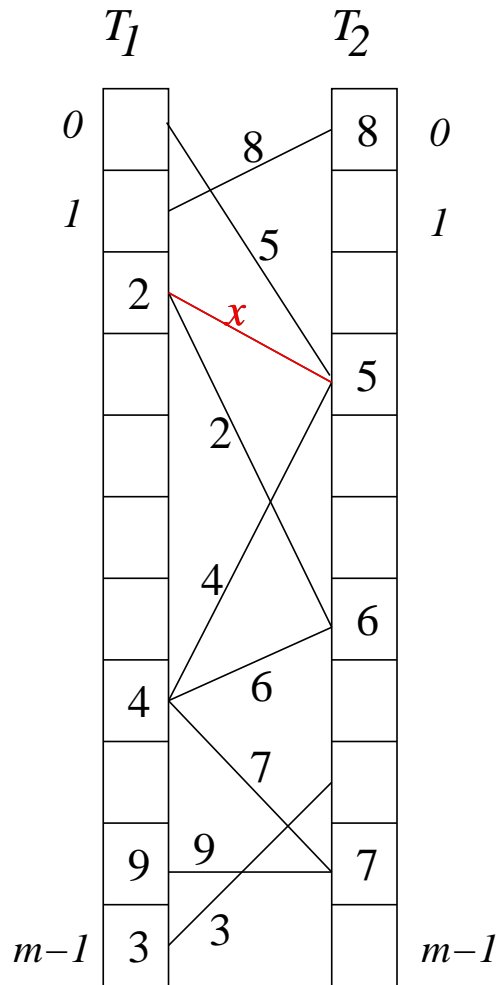


Cuckoo-Hashing



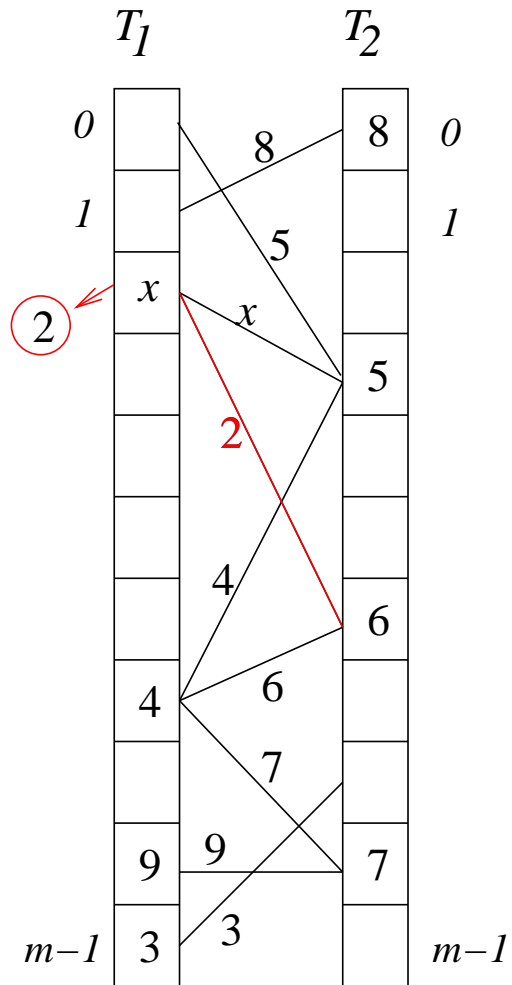
Füge x ein.

Cuckoo-Hashing



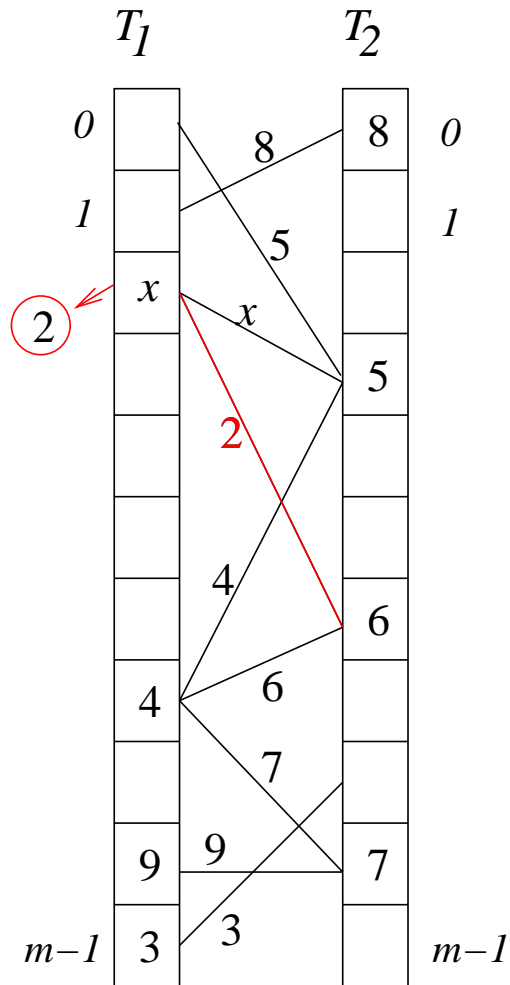
Füge x ein.
Versuche $T_1[h_1(x)]$.
Besetzt!

Cuckoo-Hashing



Schlüssel 2 aus T_1
hinauswerfen.

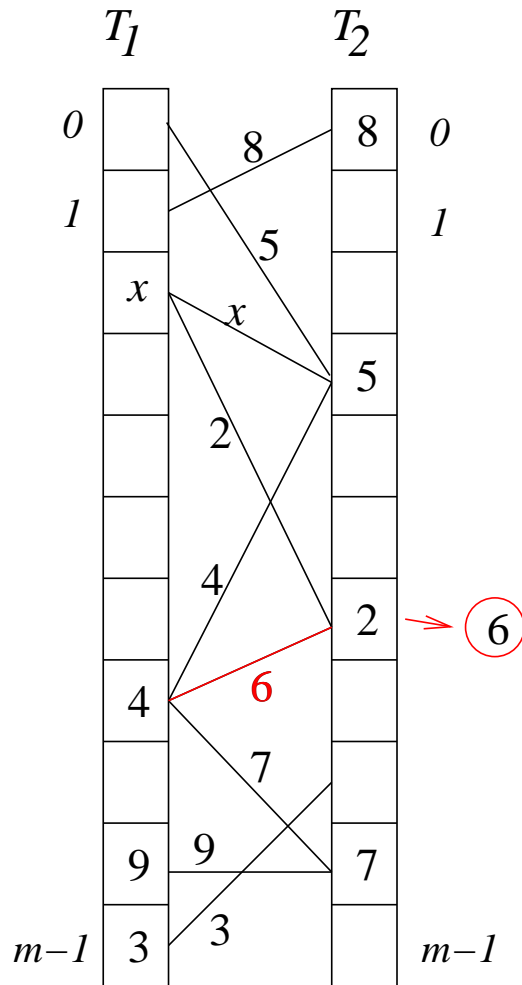
Cuckoo-Hashing



Schlüssel 2 aus T_1
hinauswerfen.

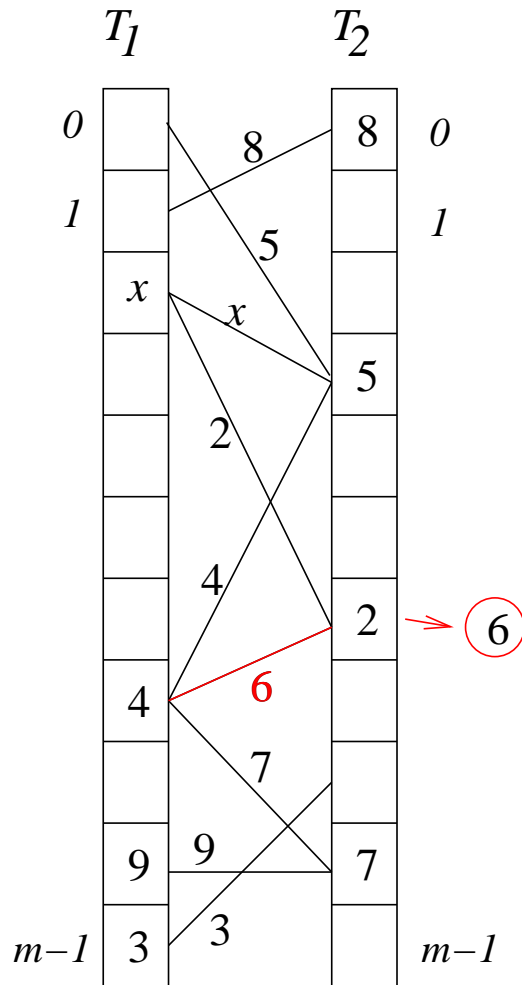
Schlüssel 2 geht zu T_2 .

Cuckoo-Hashing



Schlüssel 6 aus T_2
hinauswerfen.

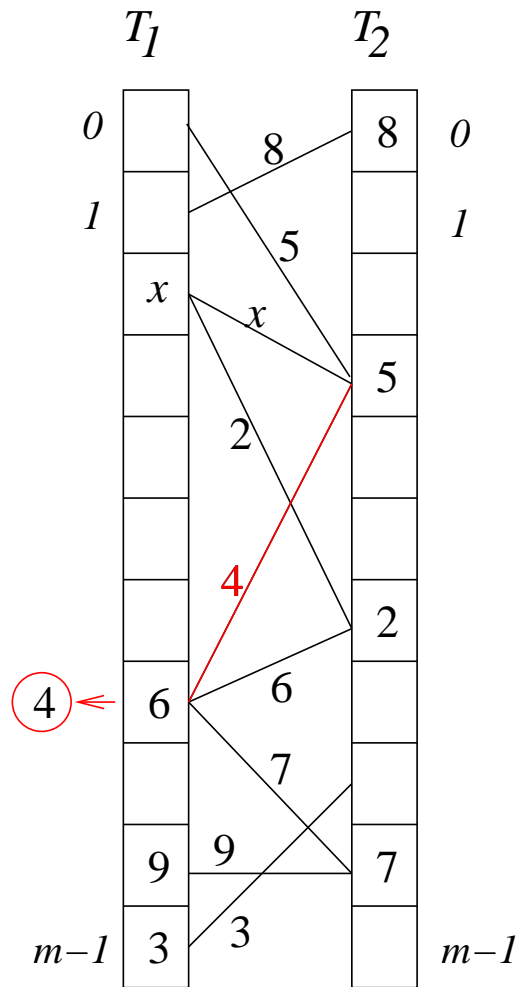
Cuckoo-Hashing



Schlüssel 6 aus T_2
hinauswerfen.

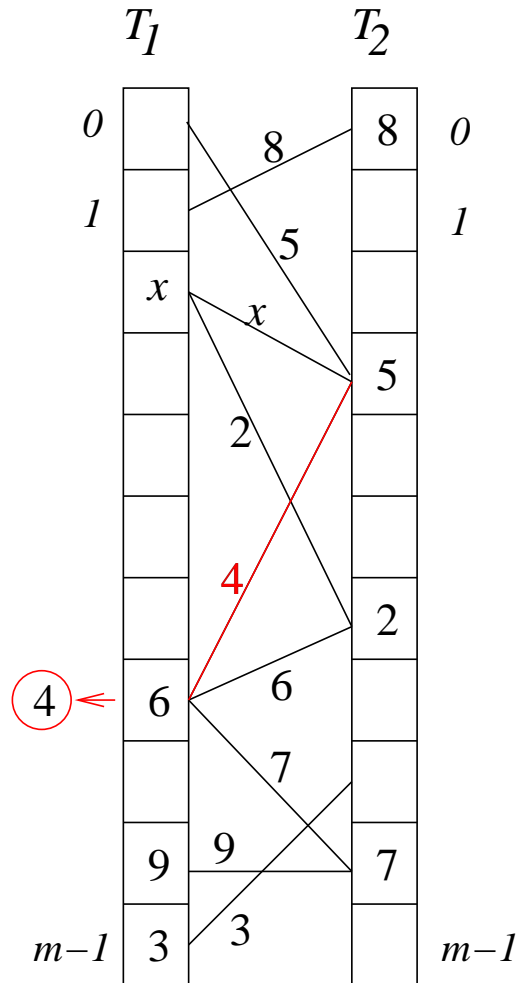
Schlüssel 6 geht zu T_1 .

Cuckoo-Hashing



Schlüssel 4 aus T_1
hinauswerfen.

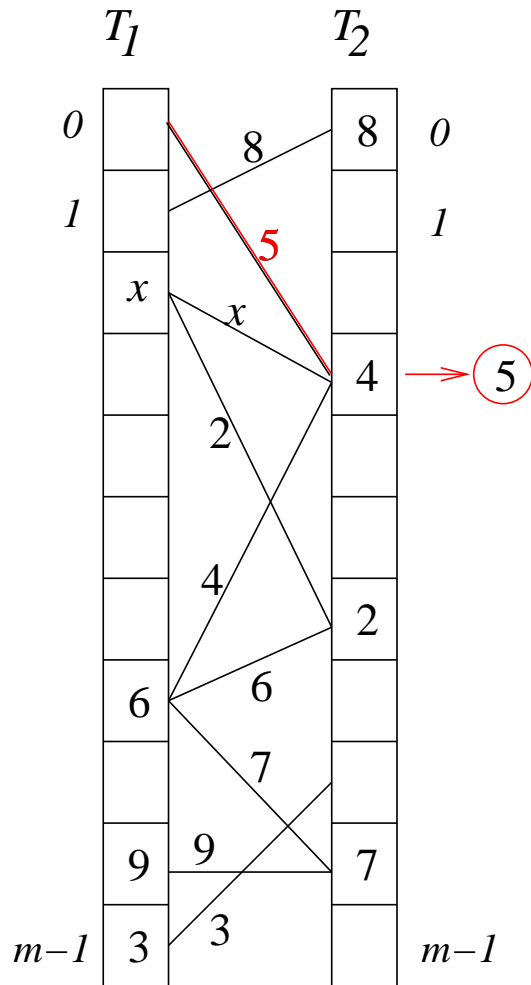
Cuckoo-Hashing



Schlüssel 4 aus T_1
hinauswerfen.

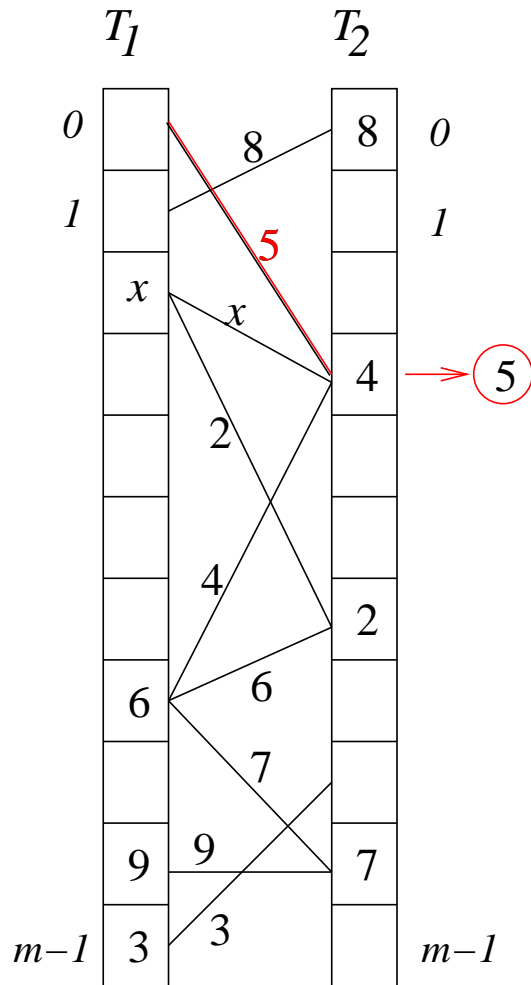
Schlüssel 4 geht zu T_2 .

Cuckoo-Hashing



Schlüssel 5 aus T_2
hinauswerfen.

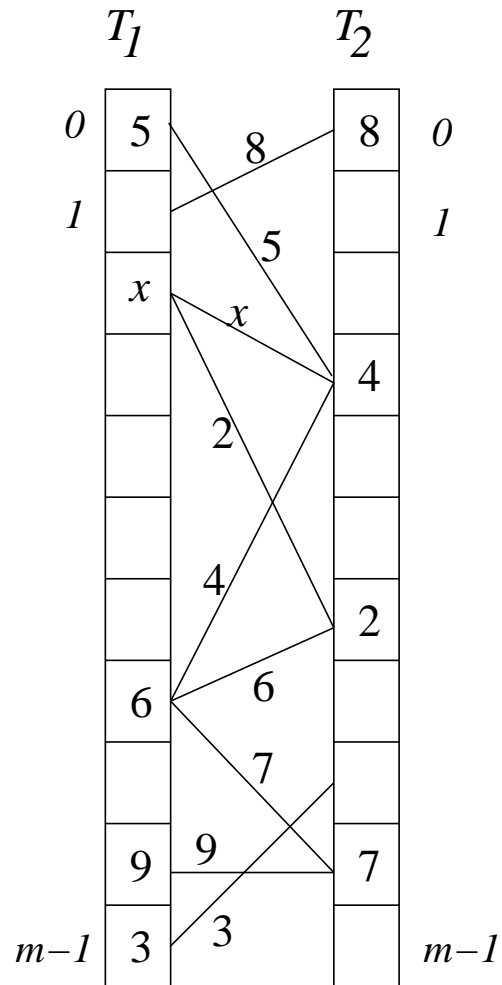
Cuckoo-Hashing



Schlüssel 5 aus T_2
hinauswerfen.

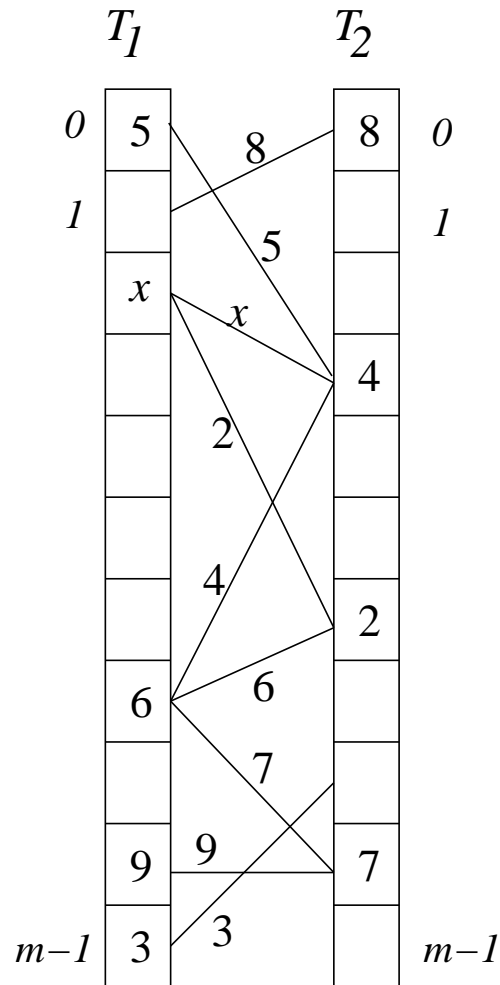
Schlüssel 5 geht zu T_1 .

Cuckoo-Hashing



Schlüssel 5 in T_1
einfügen.

Cuckoo-Hashing



Schlüssel 5 in T_1
einfügen.

Fertig!

Cuckoo-Hashing: Analyse

Wann gehen die Einfügungen gut?

Wie lange dauert eine Einfügung?

Cuckoo-Hashing: Analyse

Wann gehen die Einfügungen gut?

Wie lange dauert eine Einfügung?

Beobachtung 1 (trivial):

Wiederholte Anwendung der Einfügeprozedur platziert alle Schlüssel aus S in T_1, T_2

Cuckoo-Hashing: Analyse

Wann gehen die Einfügungen gut?

Wie lange dauert eine Einfügung?

Beobachtung 1 (trivial):

Wiederholte Anwendung der Einfügeprozedur platziert alle Schlüssel aus S in T_1, T_2

⇒

h_1, h_2 passen zu S .

Cuckoo-Hashing: Analyse

Wann gehen die Einfügungen gut?

Wie lange dauert eine Einfügung?

Beobachtung 1 (trivial):

Wiederholte Anwendung der Einfügeprozedur platziert alle Schlüssel aus S in T_1, T_2

⇒

h_1, h_2 passen zu S .

D.h.: **Wenn** h_1, h_2 **nicht** zu S passen, **dann** muss die Einfügung eines Schlüssels **fehlschlagen**.

Cuckoo-Hashing: Analyse

Wann gehen die Einfügungen gut?

Wie lange dauert eine Einfügung?

Beobachtung 1 (trivial):

Wiederholte Anwendung der Einfügeprozedur platziert alle Schlüssel aus S in T_1, T_2

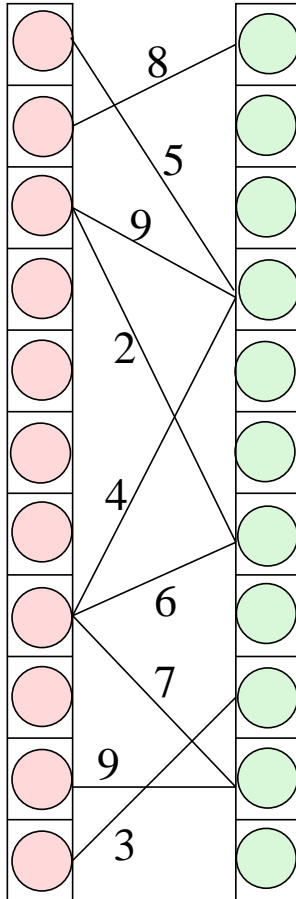
⇒

h_1, h_2 passen zu S .

D.h.: **Wenn** h_1, h_2 **nicht** zu S passen, **dann** muss die Einfügung eines Schlüssels **fehlschlagen**.

Wie sieht ein Fehlschlag aus?

Cuckoo-Hashing: Graph

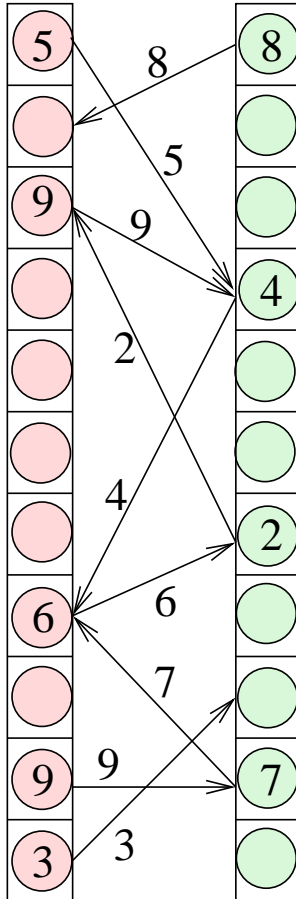


Bipartiter
Cuckoo-Graph
 $G(S, h_1, h_2)$

$$V = W = [m]$$

$$E = \{(h_1(x), h_2(x)) \mid x \in S\}$$

Cuckoo-Hashing: Graph

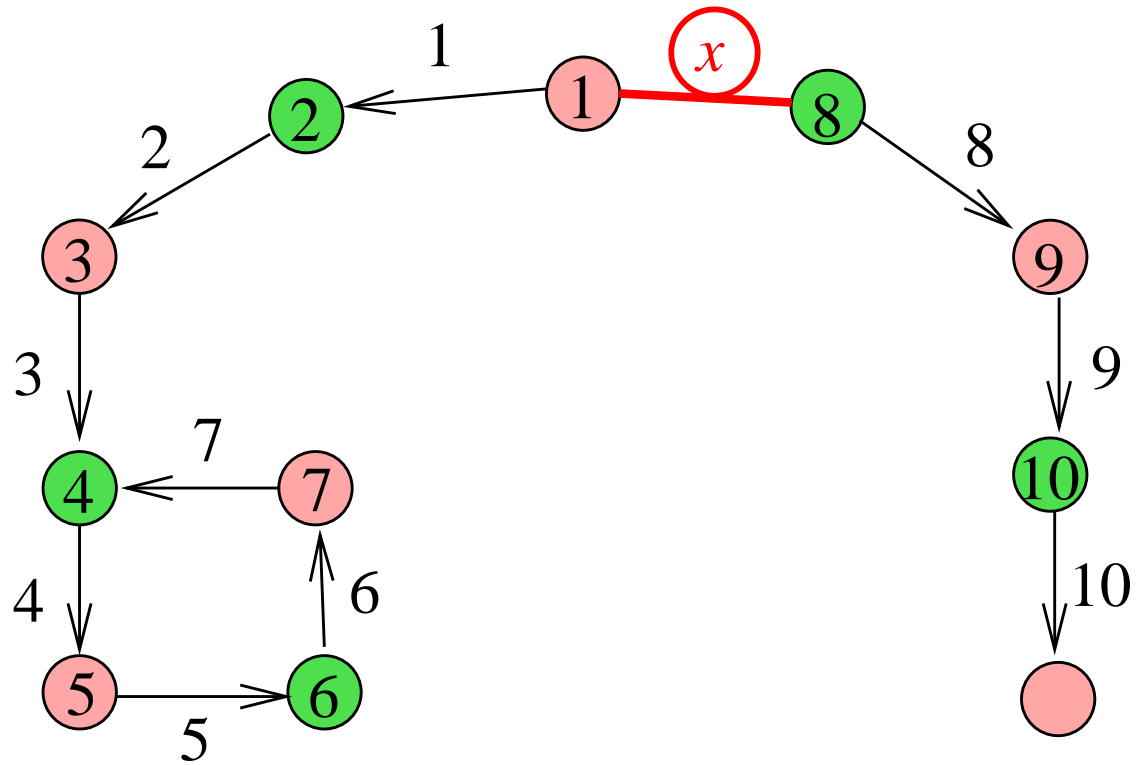


h_1, h_2 passen zu S

\Leftrightarrow

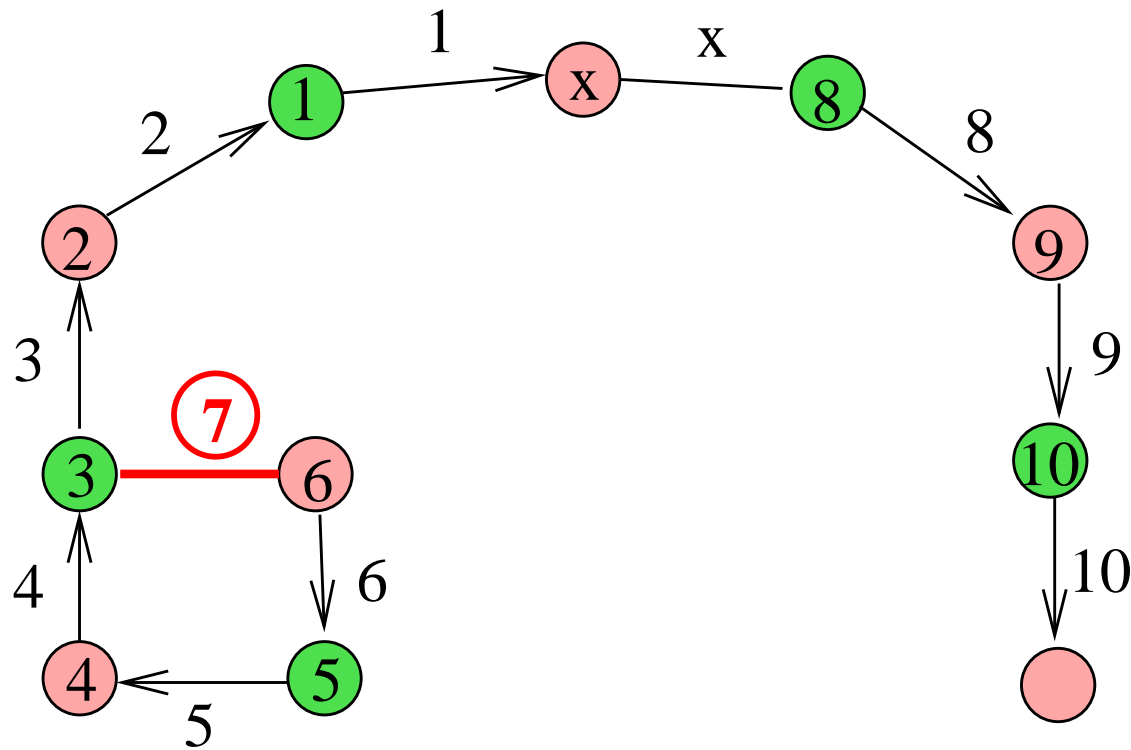
Man kann die Kanten
von $G(S, h_1, h_2)$
so richten,
dass jeder Knoten
Ausgrad ≤ 1 hat.

Cuckoo-Hashing: Fehlschlag?



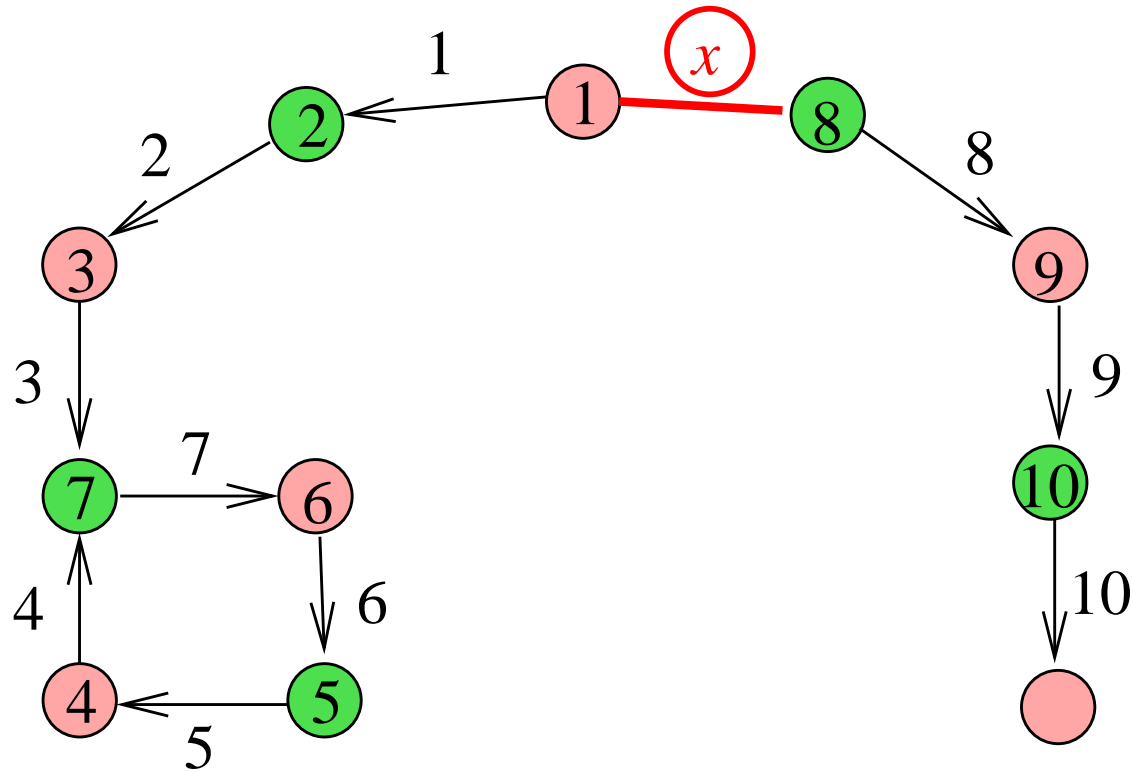
Neuer Schlüssel x .

Cuckoo-Hashing: Fehlschlag?



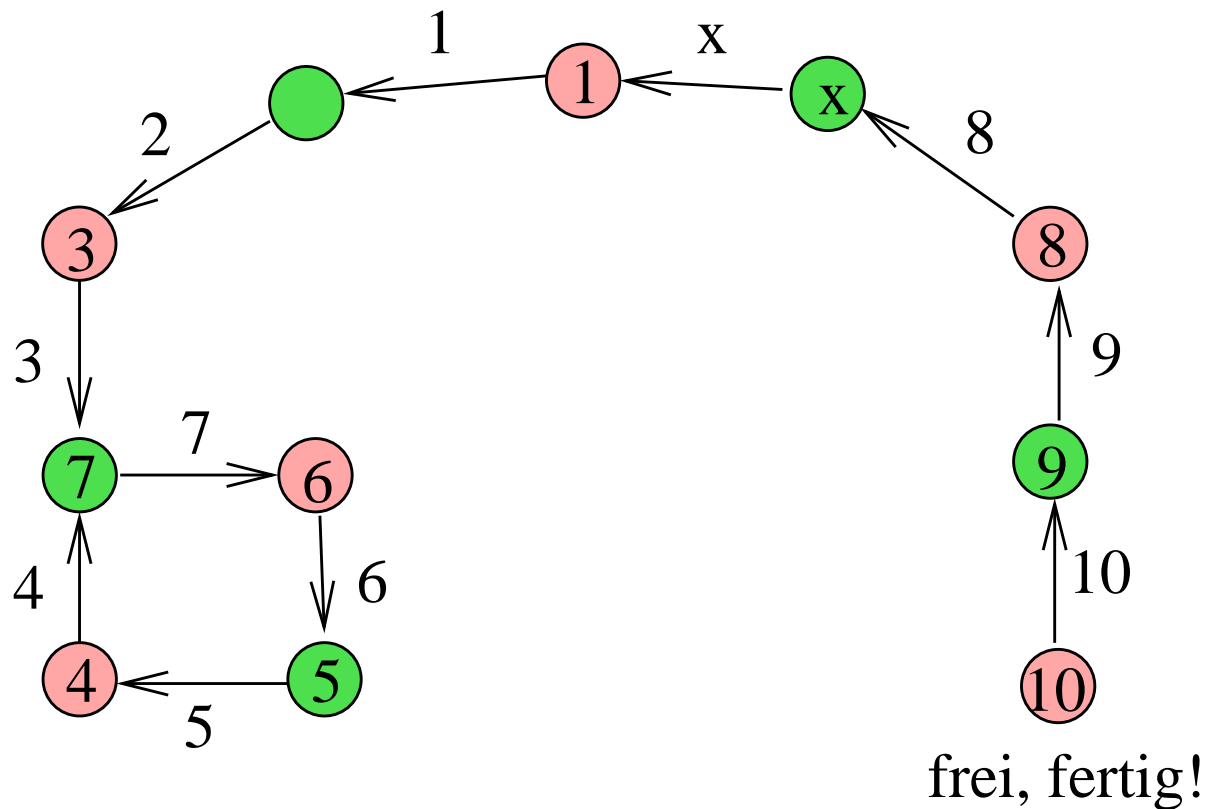
Nach 6 Runden. Schlüssel 7 ist draußen.

Cuckoo-Hashing: Fehlschlag?

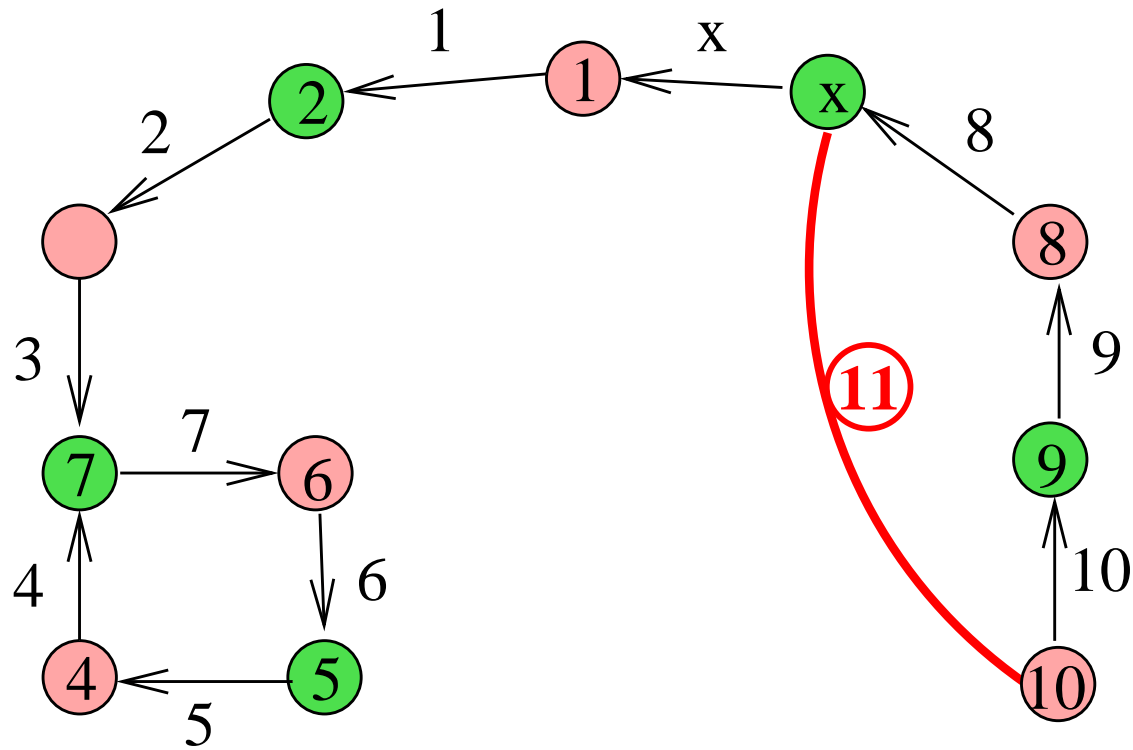


Nach weiteren 4 Runden.

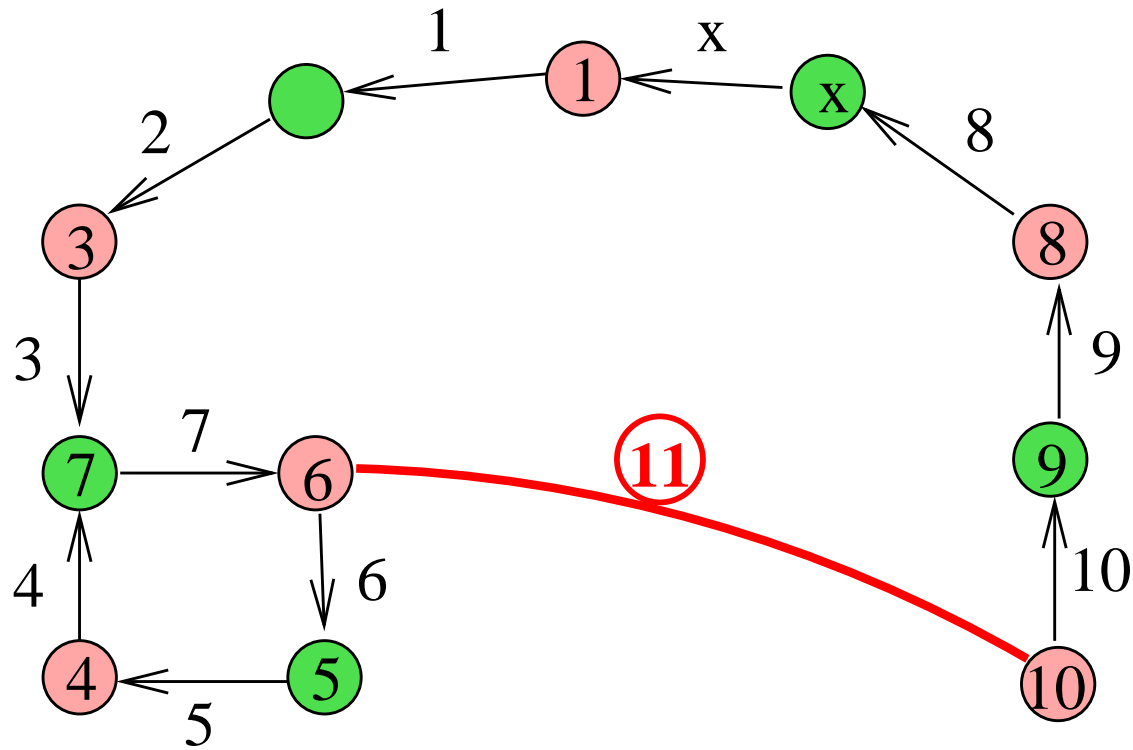
Cuckoo-Hashing: Fehlschlag?



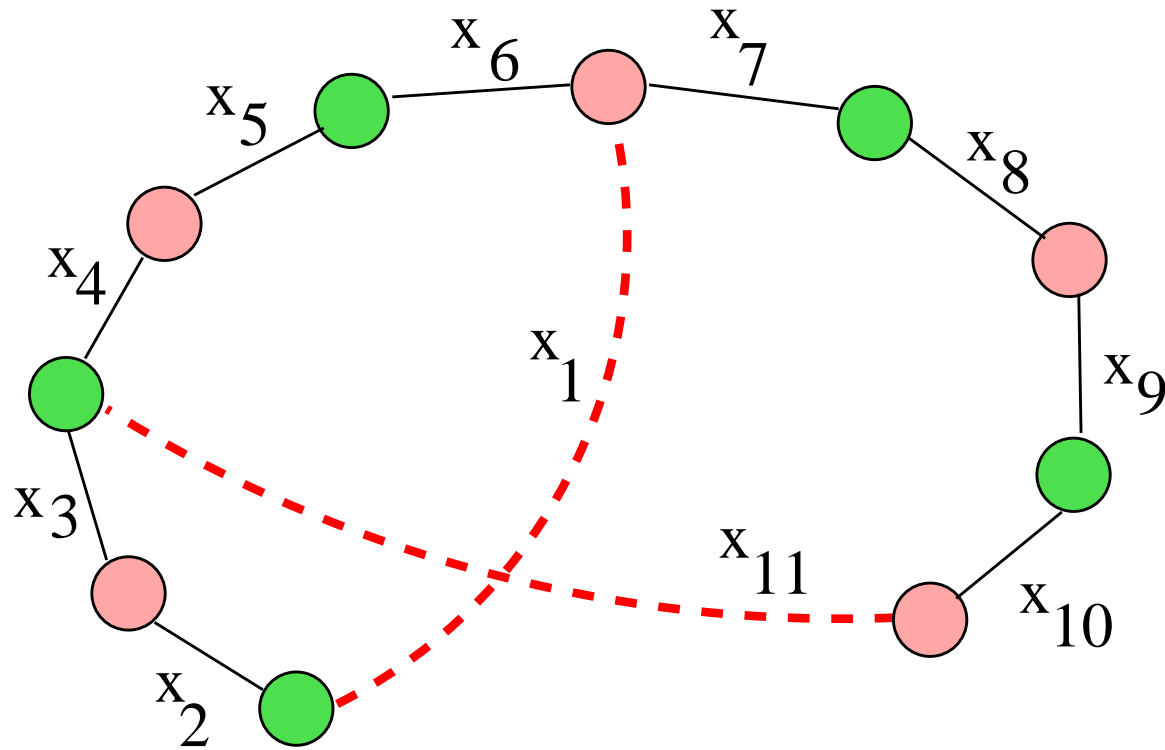
Cuckoo-Hashing: Fehlschlag!



Cuckoo-Hashing: Fehlschlag!

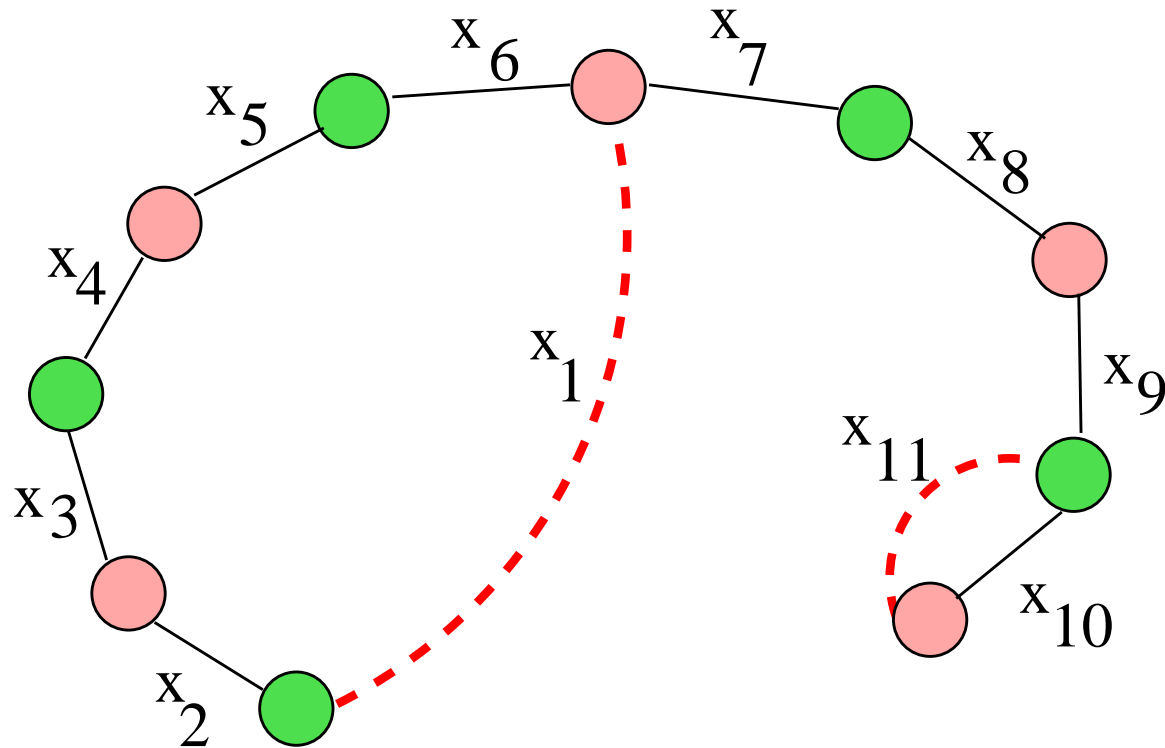


Cuckoo-Hashing: Fehlschlag



„Sperrstruktur“

Cuckoo-Hashing: Fehlschlag



„Sperrstruktur“

Cuckoo-Hashing: Analyse

Beobachtung 2: Aus dem Ablauf der Einfügung folgt:

Wenn h_1, h_2 **nicht** zu S passen, **dann** existiert in $G(S, h_1, h_2)$
eine **Sperrstruktur**,

Cuckoo-Hashing: Analyse

Beobachtung 2: Aus dem Ablauf der Einfügung folgt:

Wenn h_1, h_2 **nicht** zu S passen, **dann** existiert in $G(S, h_1, h_2)$ eine **Sperrstruktur**, d. h. es gibt eine Folge von Schlüsseln x_1, \dots, x_k in S , deren Hashwerte wie folgt kollidieren (z. B., falls k ungerade):

Cuckoo-Hashing: Analyse

Beobachtung 2: Aus dem Ablauf der Einfügung folgt:

Wenn h_1, h_2 **nicht** zu S passen, **dann** existiert in $G(S, h_1, h_2)$ eine **Sperrstruktur**, d. h. es gibt eine Folge von Schlüsseln x_1, \dots, x_k in S , deren Hashwerte wie folgt kollidieren (z. B., falls k ungerade):

$$h_2(x_1) = h_2(x_2),$$

Cuckoo-Hashing: Analyse

Beobachtung 2: Aus dem Ablauf der Einfügung folgt:

Wenn h_1, h_2 **nicht** zu S passen, **dann** existiert in $G(S, h_1, h_2)$ eine **Sperrstruktur**, d. h. es gibt eine Folge von Schlüsseln x_1, \dots, x_k in S , deren Hashwerte wie folgt kollidieren (z. B., falls k ungerade):

$$h_2(x_1) = h_2(x_2),$$

$$h_1(x_2) = h_1(x_3),$$

Cuckoo-Hashing: Analyse

Beobachtung 2: Aus dem Ablauf der Einfügung folgt:

Wenn h_1, h_2 **nicht** zu S passen, **dann** existiert in $G(S, h_1, h_2)$ eine **Sperrstruktur**, d. h. es gibt eine Folge von Schlüsseln x_1, \dots, x_k in S , deren Hashwerte wie folgt kollidieren (z. B., falls k ungerade):

$$h_2(x_1) = h_2(x_2),$$

$$h_2(x_3) = h_2(x_4),$$

$$h_1(x_2) = h_1(x_3),$$

$$h_1(x_4) = h_1(x_5),$$

Cuckoo-Hashing: Analyse

Beobachtung 2: Aus dem Ablauf der Einfügung folgt:

Wenn h_1, h_2 **nicht** zu S passen, **dann** existiert in $G(S, h_1, h_2)$ eine **Sperrstruktur**, d. h. es gibt eine Folge von Schlüsseln x_1, \dots, x_k in S , deren Hashwerte wie folgt kollidieren (z. B., falls k ungerade):

$$h_2(x_1) = h_2(x_2),$$

$$h_1(x_2) = h_1(x_3),$$

$$h_2(x_3) = h_2(x_4),$$

$$h_1(x_4) = h_1(x_5),$$

⋮

⋮

$$h_2(x_{k-2}) = h_2(x_{k-1}),$$

$$h_1(x_{k-1}) = h_1(x_k),$$

Cuckoo-Hashing: Analyse

Beobachtung 2: Aus dem Ablauf der Einfügung folgt:

Wenn h_1, h_2 **nicht** zu S passen, **dann** existiert in $G(S, h_1, h_2)$ eine **Sperrstruktur**, d. h. es gibt eine Folge von Schlüsseln x_1, \dots, x_k in S , deren Hashwerte wie folgt kollidieren (z. B., falls k ungerade):

$$h_2(x_1) = h_2(x_2),$$

$$h_1(x_2) = h_1(x_3),$$

$$h_2(x_3) = h_2(x_4),$$

$$h_1(x_4) = h_1(x_5),$$

⋮

⋮

$$h_2(x_{k-2}) = h_2(x_{k-1}),$$

$$h_1(x_{k-1}) = h_1(x_k),$$

$$h_1(x_1) \in \{h_1(x_3), \dots, h_1(x_k)\},$$

Cuckoo-Hashing: Analyse

Beobachtung 2: Aus dem Ablauf der Einfügung folgt:

Wenn h_1, h_2 **nicht** zu S passen, **dann** existiert in $G(S, h_1, h_2)$ eine **Sperrstruktur**, d. h. es gibt eine Folge von Schlüsseln x_1, \dots, x_k in S , deren Hashwerte wie folgt kollidieren (z. B., falls k ungerade):

$$h_2(x_1) = h_2(x_2),$$

$$h_1(x_2) = h_1(x_3),$$

$$h_2(x_3) = h_2(x_4),$$

$$h_1(x_4) = h_1(x_5),$$

$$\vdots$$
$$\vdots$$

$$h_2(x_{k-2}) = h_2(x_{k-1}),$$

$$h_1(x_{k-1}) = h_1(x_k),$$

$$h_1(x_1) \in \{h_1(x_3), \dots, h_1(x_k)\},$$

$$h_2(x_k) \in \{h_2(x_2), \dots, h_2(x_{k-1})\}.$$

Cuckoo-Hashing: Analyse

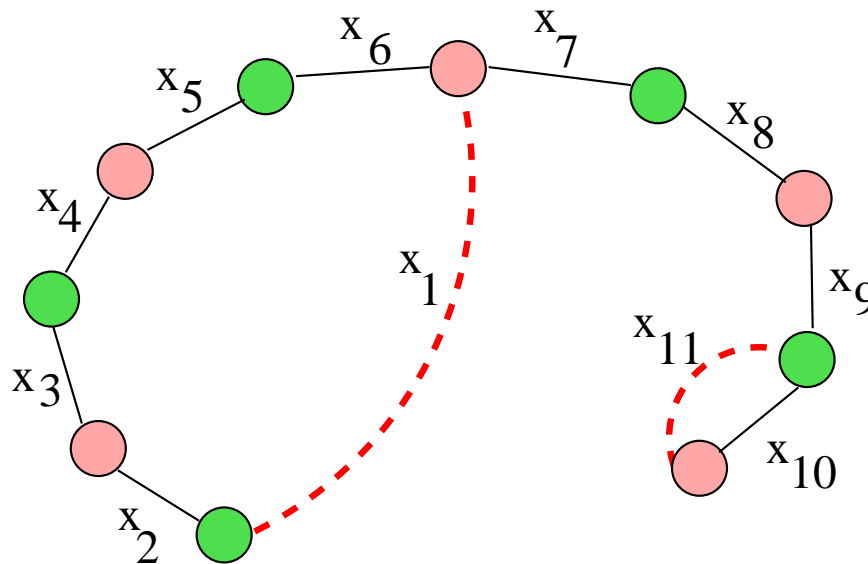
Beobachtung 2': Sogar Äquivalenz!

Wenn in $G(S, h_1, h_2)$ eine **Sperrstruktur** existiert,
dann passen h_1, h_2 nicht zu S .

Cuckoo-Hashing: Analyse

Beobachtung 2': Sogar Äquivalenz!

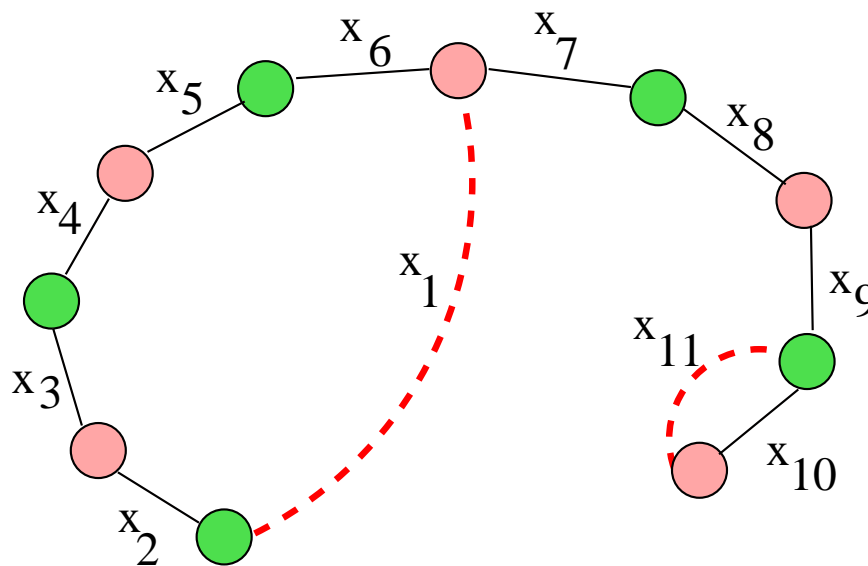
Wenn in $G(S, h_1, h_2)$ eine **Sperrstruktur** existiert, **dann** passen h_1, h_2 nicht zu S .



Cuckoo-Hashing: Analyse

Beobachtung 2': Sogar Äquivalenz!

Wenn in $G(S, h_1, h_2)$ eine **Sperrstruktur** existiert, **dann** passen h_1, h_2 nicht zu S .



Mehr Kanten/Schlüssel als Knoten/Tabellenplätze!

Cuckoo-Hashing: Analyse

Sei $n/m \leq 1 - \beta$. Dann gilt:

Cuckoo-Hashing: Analyse

Sei $n/m \leq 1 - \beta$. Dann gilt:

Pr(h_1, h_2 passen nicht zu S)

\leq

Cuckoo-Hashing: Analyse

Sei $n/m \leq 1 - \beta$. Dann gilt:

Pr(h_1, h_2 passen nicht zu S)

$$\leq \sum_{3 \leq k \leq n}$$

Cuckoo-Hashing: Analyse

Sei $n/m \leq 1 - \beta$. Dann gilt:

Pr(h_1, h_2 passen nicht zu S)

$$\leq \sum_{3 \leq k \leq n} n^k$$

Cuckoo-Hashing: Analyse

Sei $n/m \leq 1 - \beta$. Dann gilt:

Pr(h_1, h_2 passen nicht zu S)

$$\leq \sum_{3 \leq k \leq n} n^k \cdot 2 \cdot \left(\frac{1}{m}\right)^{k-1}$$

Cuckoo-Hashing: Analyse

Sei $n/m \leq 1 - \beta$. Dann gilt:

Pr(h_1, h_2 passen nicht zu S)

$$\leq \sum_{3 \leq k \leq n} n^k \cdot 2 \cdot \left(\frac{1}{m}\right)^{k-1} \cdot \left(\frac{k}{2m}\right)^2$$

Cuckoo-Hashing: Analyse

Sei $n/m \leq 1 - \beta$. Dann gilt:

Pr(h_1, h_2 passen nicht zu S)

$$\leq \sum_{3 \leq k \leq n} n^k \cdot 2 \cdot \left(\frac{1}{m}\right)^{k-1} \cdot \left(\frac{k}{2m}\right)^2$$

<

Cuckoo-Hashing: Analyse

Sei $n/m \leq 1 - \beta$. Dann gilt:

Pr(h_1, h_2 passen nicht zu S)

$$\begin{aligned} &\leq \sum_{3 \leq k \leq n} n^k \cdot 2 \cdot \left(\frac{1}{m}\right)^{k-1} \cdot \left(\frac{k}{2m}\right)^2 \\ &< \frac{1}{2m} \cdot \sum_{k \geq 3} \end{aligned}$$

Cuckoo-Hashing: Analyse

Sei $n/m \leq 1 - \beta$. Dann gilt:

Pr(h_1, h_2 passen nicht zu S)

$$\begin{aligned} &\leq \sum_{3 \leq k \leq n} n^k \cdot 2 \cdot \left(\frac{1}{m}\right)^{k-1} \cdot \left(\frac{k}{2m}\right)^2 \\ &< \frac{1}{2m} \cdot \sum_{k \geq 3} k^2 \cdot \left(\frac{n}{m}\right)^k \end{aligned}$$

Cuckoo-Hashing: Analyse

Sei $n/m \leq 1 - \beta$. Dann gilt:

Pr(h_1, h_2 passen nicht zu S)

$$\leq \sum_{3 \leq k \leq n} n^k \cdot 2 \cdot \left(\frac{1}{m}\right)^{k-1} \cdot \left(\frac{k}{2m}\right)^2$$

$$< \frac{1}{2m} \cdot \sum_{k \geq 3} k^2 \cdot \left(\frac{n}{m}\right)^k$$

$$\leq$$

Cuckoo-Hashing: Analyse

Sei $n/m \leq 1 - \beta$. Dann gilt:

Pr(h_1, h_2 passen nicht zu S)

$$\begin{aligned} &\leq \sum_{3 \leq k \leq n} n^k \cdot 2 \cdot \left(\frac{1}{m}\right)^{k-1} \cdot \left(\frac{k}{2m}\right)^2 \\ &< \frac{1}{2m} \cdot \sum_{k \geq 3} k^2 \cdot \left(\frac{n}{m}\right)^k \\ &\leq \frac{1}{2m} \cdot \sum_{k \geq 3} \end{aligned}$$

Cuckoo-Hashing: Analyse

Sei $n/m \leq 1 - \beta$. Dann gilt:

Pr(h_1, h_2 passen nicht zu S)

$$\begin{aligned} &\leq \sum_{3 \leq k \leq n} n^k \cdot 2 \cdot \left(\frac{1}{m}\right)^{k-1} \cdot \left(\frac{k}{2m}\right)^2 \\ &< \frac{1}{2m} \cdot \sum_{k \geq 3} k^2 \cdot \left(\frac{n}{m}\right)^k \\ &\leq \frac{1}{2m} \cdot \sum_{k \geq 3} k^2 \cdot (1 - \beta)^k \end{aligned}$$

Cuckoo-Hashing: Analyse

Sei $n/m \leq 1 - \beta$. Dann gilt:

Pr(h_1, h_2 passen nicht zu S)

$$\begin{aligned} &\leq \sum_{3 \leq k \leq n} n^k \cdot 2 \cdot \left(\frac{1}{m}\right)^{k-1} \cdot \left(\frac{k}{2m}\right)^2 \\ &< \frac{1}{2m} \cdot \sum_{k \geq 3} k^2 \cdot \left(\frac{n}{m}\right)^k \\ &\leq \frac{1}{2m} \cdot \sum_{k \geq 3} k^2 \cdot (1 - \beta)^k = O\left(\frac{1}{\beta^3 \cdot m}\right). \end{aligned}$$

Cuckoo-Hashing: Analyse

(Es gibt $\leq n^k$ Folgen (x_1, \dots, x_k) von Schlüsseln.

Die erste Kante kann einen h_1 - oder einen h_2 -Endpunkt haben.

Dass die Hashwerte entlang des Weges übereinstimmen, hat Wahrscheinlichkeit $1/m^{k-1}$.

Die Anzahl der möglichen Trefferknoten für jede der beiden „zurückschlagenden“ Kanten ist $\leq k/2$.)

Cuckoo-Hashing: Analyse

Benötige: Mechanismus, um Endlosschleifen (Einfügung, die Sperrstruktur erzeugt) abzurechnen.

Cuckoo-Hashing: Analyse

Benötige: Mechanismus, um Endlosschleifen (Einfügung, die Sperrstruktur erzeugt) abzurechnen.

Lasse Einfügezyklus für **maximal** $C \log n$ Schritte laufen, C eine Konstante.

Danach: „Rehash“ (neue Hashfunktion, Neuaufbau).

Cuckoo-Hashing: Analyse

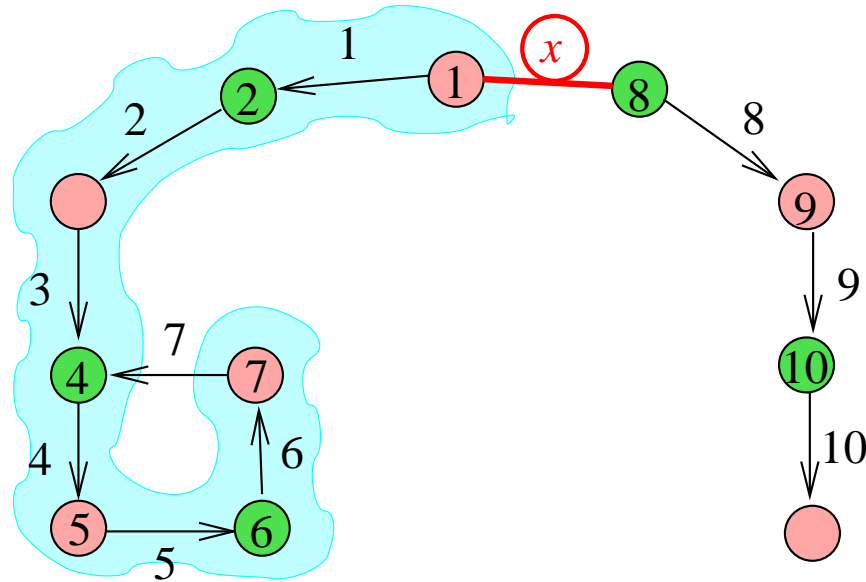
Benötige: Mechanismus, um Endlosschleifen (Einfügung, die Sperrstruktur erzeugt) abzurechnen.

Lasse Einfügezyklus für **maximal** $C \log n$ Schritte laufen, C eine Konstante.

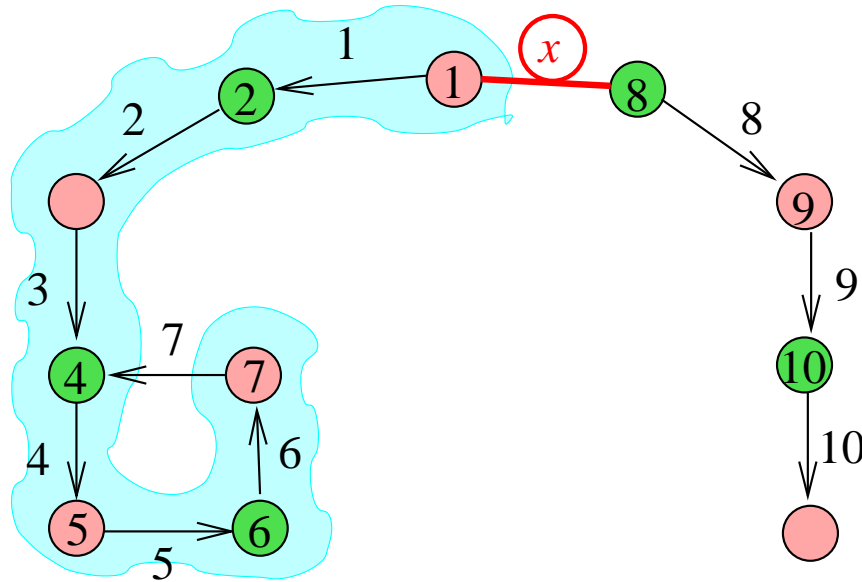
Danach: „Rehash“ (neue Hashfunktion, Neuaufbau).

Will aber keine erfolgreiche Einfügung verlieren, nur weil sie lange dauert!

Cuckoo-Hashing: Analyse



Cuckoo-Hashing: Analyse



Beobachtung 3:

Wenn die **erfolgreiche** Einfügung von x mindestens t Verdrängungen durchführt, dann gibt es in $G(S, h_1, h_2)$ einen einfachen Weg der Länge $\geq \lceil (t-1)/3 \rceil$, der an Knoten $h_1(x)$ oder $h_2(x)$ beginnt.

Cuckoo-Hashing: Analyse

Pr($\geq t$ Verdrängungsschritte)

Cuckoo-Hashing: Analyse

Pr($\geq t$ Verdrängungsschritte)

$$\leq 2 \cdot n^{\lceil (t-1)/3 \rceil} \left(\frac{1}{m} \right)^{\lceil (t-1)/3 \rceil}$$

$$\leq 2 \cdot (1 - \beta)^{(t-1)/3}$$

$$\leq 2^{c-t/d}, \text{ für Konstante } c, d.$$

Cuckoo-Hashing: Analyse

Pr($\geq t$ Verdrängungsschritte)

$$\leq 2 \cdot n^{\lceil (t-1)/3 \rceil} \left(\frac{1}{m} \right)^{\lceil (t-1)/3 \rceil}$$

$$\leq 2 \cdot (1 - \beta)^{(t-1)/3}$$

$$\leq 2^{c-t/d}, \text{ für Konstante } c, d.$$

\Rightarrow **Pr**($\geq C \log n$ Verdrängungsschritte für x) = $O(1/m^3)$
für Konstante $C \geq \frac{3 \ln 2}{\beta}$ [$\geq 3 / \log(1/(1 - \beta))$].

Cuckoo-Hashing: Analyse

Pr($\geq t$ Verdrängungsschritte)

$$\leq 2 \cdot n^{\lceil (t-1)/3 \rceil} \left(\frac{1}{m} \right)^{\lceil (t-1)/3 \rceil}$$

$$\leq 2 \cdot (1 - \beta)^{(t-1)/3}$$

$$\leq 2^{c-t/d}, \text{ für Konstante } c, d.$$

\Rightarrow **Pr**($\geq C \log n$ Verdrängungsschritte für x) = $O(1/m^3)$
für Konstante $C \geq \frac{3 \ln 2}{\beta}$ [$\geq 3 / \log(1/(1 - \beta))$].

\Rightarrow **Pr**($\exists x \in S$: $\geq C \log n$ Verdrängungsschritte für x) =
 $O(1/m^2)$.

Cuckoo-Hashing: Analyse

Einfügung von x kostet:

$E(\text{Anzahl Verdrängungsschritte})$

* Standardformel für Erwartungswerte.

Cuckoo-Hashing: Analyse

Einfügung von x kostet:

\mathbf{E} (Anzahl Verdrängungsschritte)

$$\leq \sum_{t \geq 1} \mathbf{Pr}(\geq t \text{ Verdrängungsschritte})$$

* Standardformel für Erwartungswerte.

Cuckoo-Hashing: Analyse

Einfügung von x kostet:

\mathbf{E} (Anzahl Verdrängungsschritte)

$$\begin{aligned} &\stackrel{*}{\leq} \sum_{t \geq 1} \mathbf{Pr}(\geq t \text{ Verdrängungsschritte}) \\ &\leq \sum_{t \geq 1} 2^{c-t/d} \\ &= O(1). \end{aligned}$$

* Standardformel für Erwartungswerte.

Cuckoo-Hashing: Zusammenfassung

Mitteilung 0.4.2

Vorausgesetzt: Hashfunktionen verteilen Schlüssel rein zufällig.

Wenn man Cuckoo-Hashing mit n Schlüsseln in zwei Tabellen mit je m Plätzen durchführt, wobei

$$n/m \leq 1 - \beta \quad (\text{essenziell!!}),$$

dann ist die Wahrscheinlichkeit dafür, dass h_1, h_2 zur Schlüsselmenge passen, $1 - O(1/m)$.

Cuckoo-Hashing: Zusammenfassung

Mitteilung 0.4.2

Vorausgesetzt: Hashfunktionen verteilen Schlüssel rein zufällig.

Wenn man Cuckoo-Hashing mit n Schlüsseln in zwei Tabellen mit je m Plätzen durchführt, wobei

$$n/m \leq 1 - \beta \quad (\text{essenziell!!}),$$

dann ist die Wahrscheinlichkeit dafür, dass h_1, h_2 zur Schlüsselmenge passen, $1 - O(1/m)$.

Falls h_1, h_2 passen, ist die erwartete Einfügezeit $O(1/\beta^3)$.

Cuckoo-Hashing: Zusammenfassung

Bei Auftreten eines Fehlers

(z. B. Einfügung nach $10 \log n$ Runden nicht beendet):

„Rehash“ (neue Hashfunktionen wählen)

mit erwarteten Kosten $O(n)$.

Cuckoo-Hashing: Zusammenfassung

Bei Auftreten eines Fehlers

(z. B. Einfügung nach $10 \log n$ Runden nicht beendet):

„Rehash“ (neue Hashfunktionen wählen)

mit erwarteten Kosten $O(n)$.

„Amortisierte“ erwartete Einfügezeit: $O(1)$.

Cuckoo-Hashing: Zusammenfassung

Bei Auftreten eines Fehlers

(z. B. Einfügung nach $10 \log n$ Runden nicht beendet):

„Rehash“ (neue Hashfunktionen wählen)

mit erwarteten Kosten $O(n)$.

„Amortisierte“ erwartete Einfügezeit: $O(1)$.

Auslastungsfaktor: $\frac{1}{2}(1 - \beta) < \frac{1}{2}$.

Cuckoo-Hashing: Zusammenfassung

Bei Auftreten eines Fehlers

(z. B. Einfügung nach $10 \log n$ Runden nicht beendet):

„Rehash“ (neue Hashfunktionen wählen)

mit erwarteten Kosten $O(n)$.

„Amortisierte“ erwartete Einfügezeit: $O(1)$.

Auslastungsfaktor: $\frac{1}{2}(1 - \beta) < \frac{1}{2}$.

Nachteil: Auslastung der Tabelle unter 50%.

Cuckoo-Hashing: Zusammenfassung

Bei Auftreten eines Fehlers

(z. B. Einfügung nach $10 \log n$ Runden nicht beendet):

„Rehash“ (neue Hashfunktionen wählen)

mit erwarteten Kosten $O(n)$.

„Amortisierte“ erwartete Einfügezeit: $O(1)$.

Auslastungsfaktor: $\frac{1}{2}(1 - \beta) < \frac{1}{2}$.

Nachteil: Auslastung der Tabelle unter 50%.

Abhilfe: Mehr als 2 Hashfunktionen oder größere Behälter.

(Details: später.)

Cuckoo-Hashing: Zusammenfassung

Bei Auftreten eines Fehlers

(z. B. Einfügung nach $10 \log n$ Runden nicht beendet):

„Rehash“ (neue Hashfunktionen wählen)

mit erwarteten Kosten $O(n)$.

„Amortisierte“ erwartete Einfügezeit: $O(1)$.

Auslastungsfaktor: $\frac{1}{2}(1 - \beta) < \frac{1}{2}$.

Nachteil: Auslastung der Tabelle unter 50%.

Abhilfe: Mehr als 2 Hashfunktionen oder größere Behälter.

(Details: später.)

Praktisch sehr effizient (auch mit Tabellenhashing).

Multiplikative/lineare Hashfunktionen nicht benutzen!