SS 2021

Algorithmen und Datenstrukturen

5. Kapitel

Hashverfahren

Martin Dietzfelbinger

Mai 2021

5.1 Grundbegriffe

Gegeben: Universum U (auch *ohne Ordnung*) und Wertebereich R.

5.1 Grundbegriffe

Gegeben: Universum U (auch *ohne Ordnung*) und Wertebereich R.

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

5.1 Grundbegriffe

Gegeben: Universum U (auch *ohne Ordnung*) und Wertebereich R.

Hashverfahren (oder **Schlüsseltransformationsverfahren**) implementieren ein **Wörterbuch** (alias **dynamische Abbildung**, **assoziatives Array**)

5.1 Grundbegriffe

Gegeben: Universum U (auch *ohne Ordnung*) und Wertebereich R.

Hashverfahren (oder **Schlüsseltransformationsverfahren**) implementieren ein **Wörterbuch** (alias **dynamische Abbildung**, **assoziatives Array**)

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

5.1 Grundbegriffe

Gegeben: Universum U (auch *ohne Ordnung*) und Wertebereich R.

Hashverfahren (oder **Schlüsseltransformationsverfahren**) implementieren ein **Wörterbuch** (alias **dynamische Abbildung**, **assoziatives Array**)

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

Ziel:

5.1 Grundbegriffe

Gegeben: Universum U (auch *ohne Ordnung*) und Wertebereich R.

Hashverfahren (oder **Schlüsseltransformationsverfahren**) implementieren ein **Wörterbuch** (alias **dynamische Abbildung**, **assoziatives Array**)

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

Ziel:

5.1 Grundbegriffe

Gegeben: Universum U (auch *ohne Ordnung*) und Wertebereich R.

Hashverfahren (oder **Schlüsseltransformationsverfahren**) implementieren ein **Wörterbuch** (alias **dynamische Abbildung**, **assoziatives Array**)

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

Ziel:

Zeit $O(1)$ pro Operation

5.1 Grundbegriffe

Gegeben: Universum U (auch *ohne Ordnung*) und Wertebereich R.

Hashverfahren (oder **Schlüsseltransformationsverfahren**) implementieren ein **Wörterbuch** (alias **dynamische Abbildung**, **assoziatives Array**)

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

Ziel:

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

5.1 Grundbegriffe

Gegeben: Universum U (auch *ohne Ordnung*) und Wertebereich R.

Hashverfahren (oder **Schlüsseltransformationsverfahren**) implementieren ein **Wörterbuch** (alias **dynamische Abbildung**, **assoziatives Array**)

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

Ziel:

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

5.1 Grundbegriffe

Gegeben: Universum U (auch *ohne Ordnung*) und Wertebereich R.

Hashverfahren (oder **Schlüsseltransformationsverfahren**) implementieren ein **Wörterbuch** (alias **dynamische Abbildung**, **assoziatives Array**)

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

Ziel:

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

5.1 Grundbegriffe

Gegeben: Universum U (auch *ohne Ordnung*) und Wertebereich R.

Hashverfahren (oder **Schlüsseltransformationsverfahren**) implementieren ein **Wörterbuch** (alias **dynamische Abbildung**, **assoziatives Array**)

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

Ziel:

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

Gegensatz **Suchbäume:**

5.1 Grundbegriffe

Gegeben: Universum U (auch *ohne Ordnung*) und Wertebereich R.

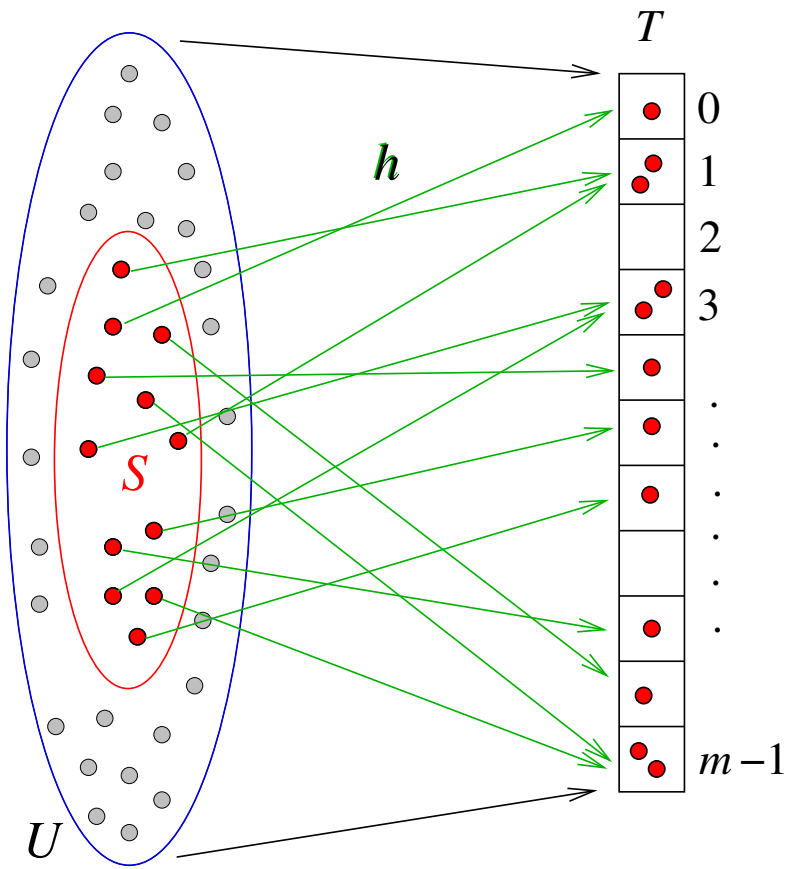
Hashverfahren (oder **Schlüsseltransformationsverfahren**) implementieren ein **Wörterbuch** (alias **dynamische Abbildung**, **assoziatives Array**)

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

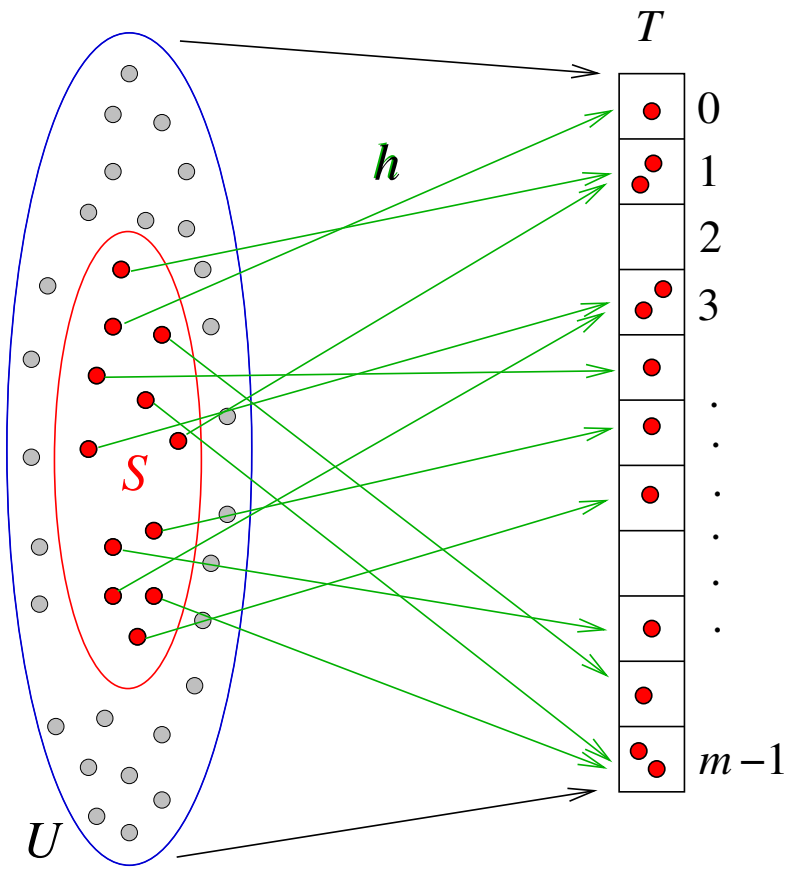
Ziel:

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

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

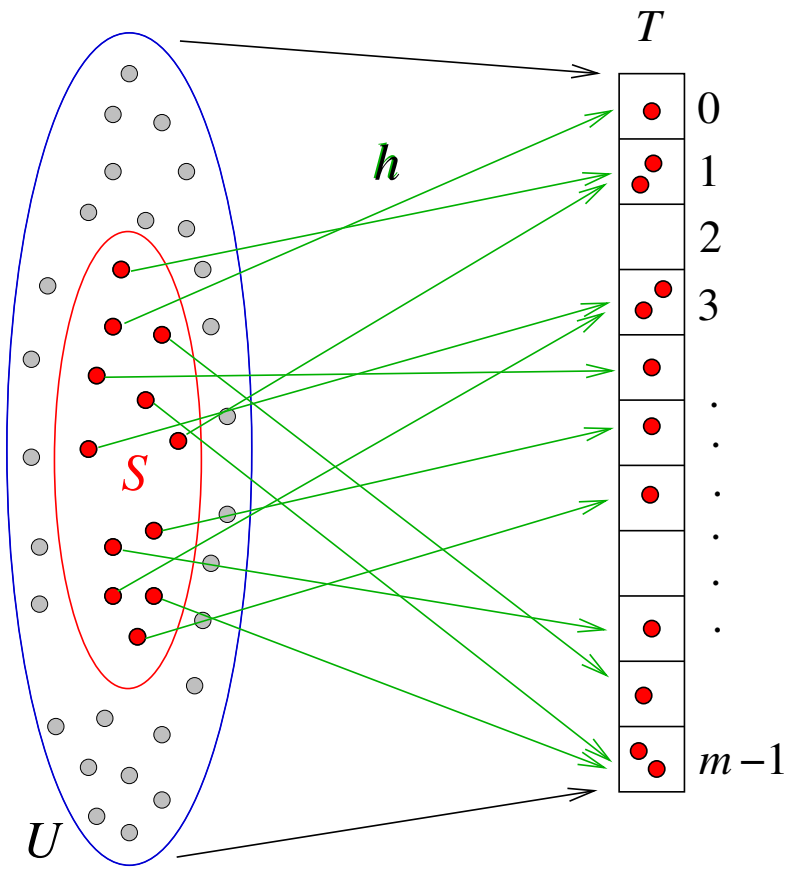


* to hash (*engl.*): kleinhacken, kleinschneiden



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

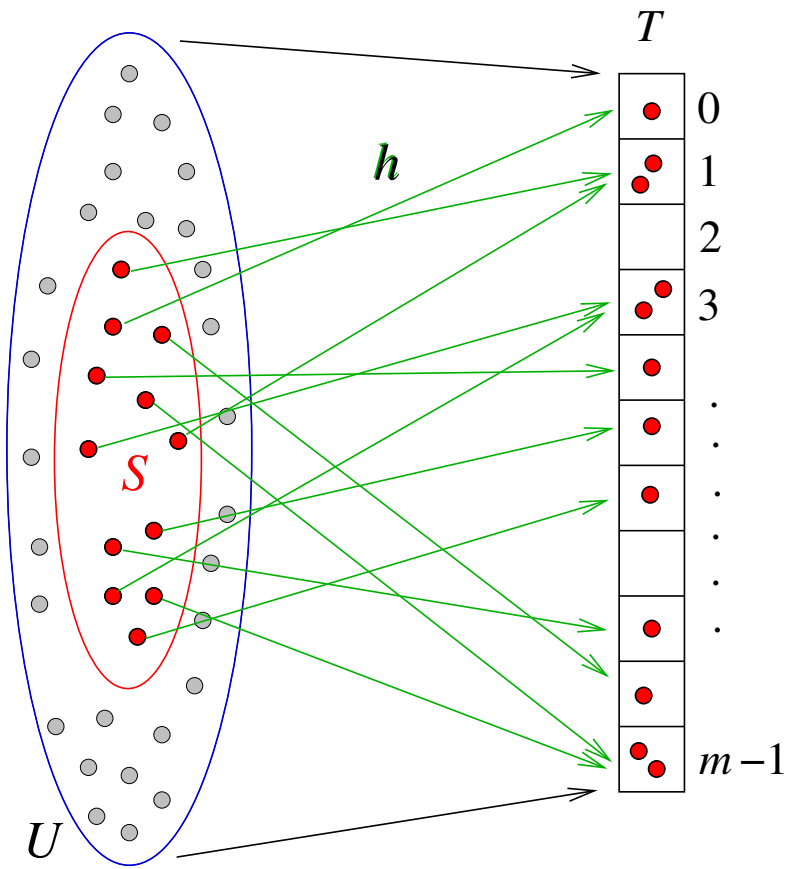
* to hash (*engl.*): kleinhacken, kleinschneiden



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

U : Universum der Schlüssel

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

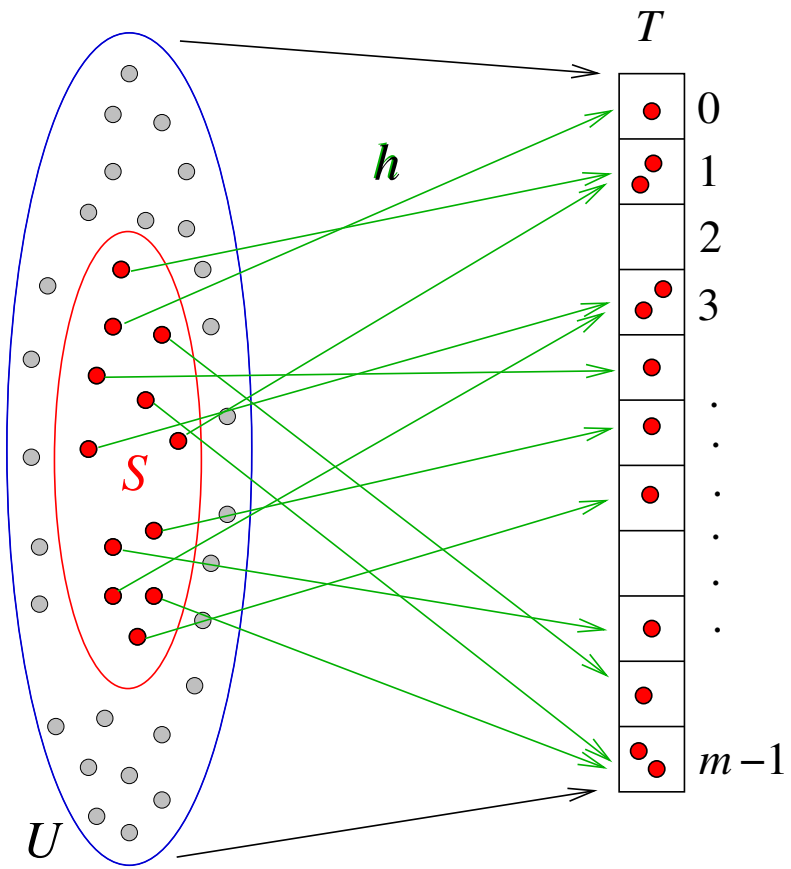


Hashfunktion* $h: U \to [m]$

U : Universum der Schlüssel

$[m] = \{0, \ldots, m-1\}$:
Indexbereich für Tabelle T

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



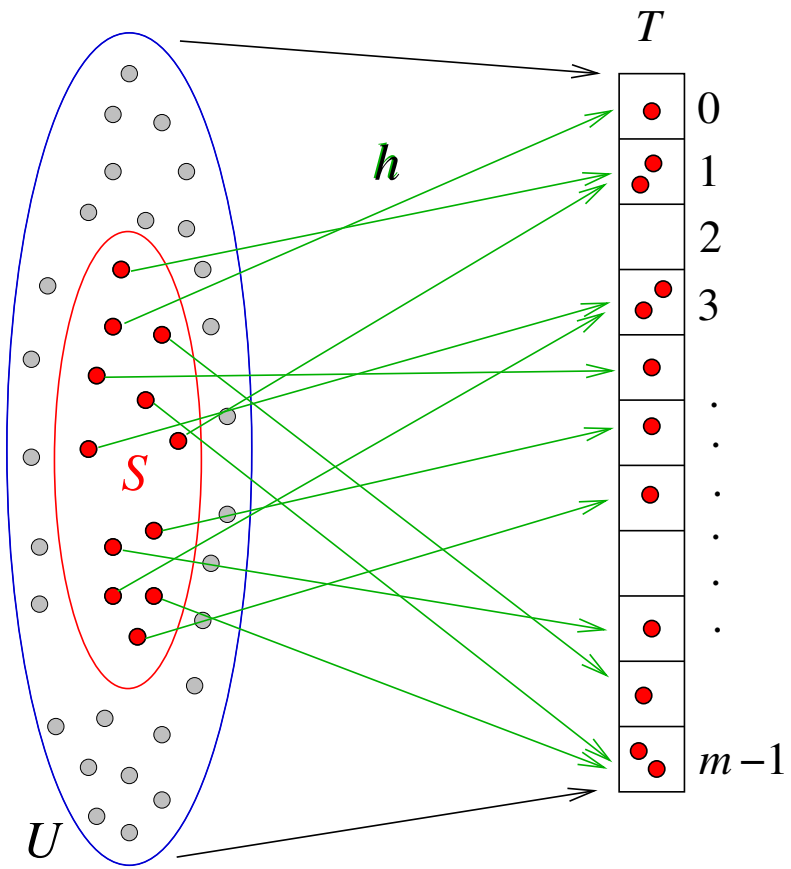
Hashfunktion* $h: U \to [m]$

U : Universum der Schlüssel

$[m] = \{0, \ldots, m-1\}$:
Indexbereich für Tabelle T

$S \subseteq U$

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



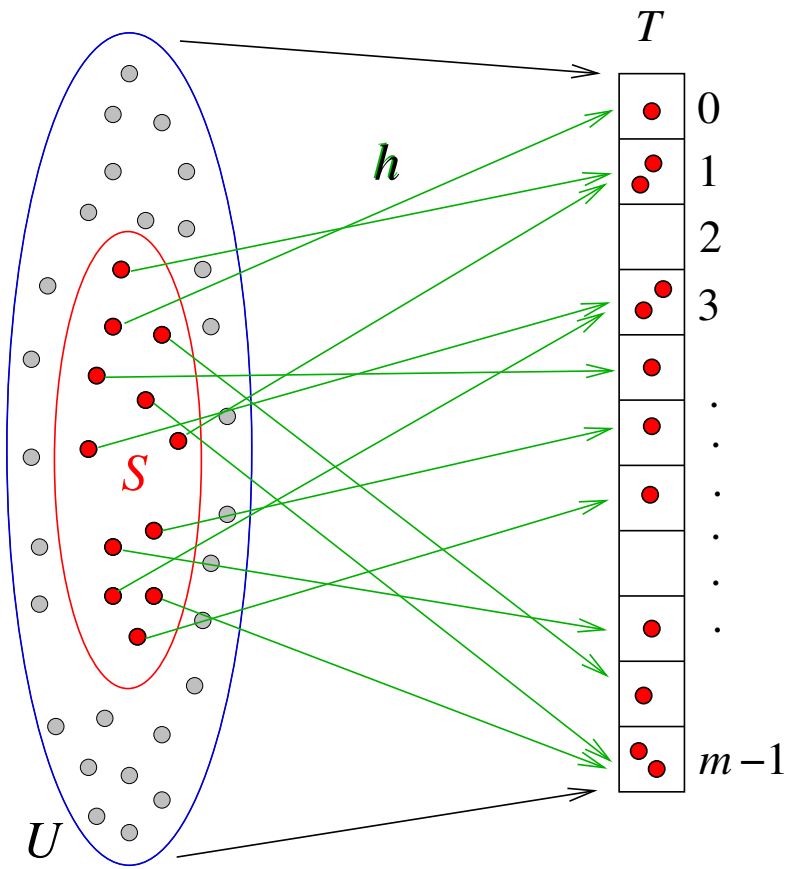
Hashfunktion* $h: U \to [m]$

U : Universum der Schlüssel

$[m] = \{0, \ldots, m-1\}$:
Indexbereich für Tabelle T

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

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



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

U : Universum der Schlüssel

$[m] = \{0, \ldots, m-1\}$:
Indexbereich für Tabelle T

$S \subseteq U$, $|S| = n$
Definitionsbereich von f.

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

Grundansatz:

Grundansatz:

Speicherung von $f\colon S \to R$ in Tabelle $\texttt{T}[0\,..\,m-1]$.

Grundansatz:

Speicherung von $f\colon S \to R$ in Tabelle $\texttt{T}[0\,..\,m-1]$.

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

Grundansatz:

Speicherung von $f\colon S \to R$ in Tabelle $\texttt{T}[0\,..\,m-1]$.

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

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

Grundansatz:

Speicherung von $f\colon S \to R$ in 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[`$h(x)$`]`.

Beispiel: $U = \{\mathtt{A}, \ldots, \mathtt{Z}, \mathtt{a}, \ldots, \mathtt{z}\}^{3..20}$

* $x \bmod m$: Rest bei der Division von x durch m.

Beispiel: $U = \{\mathtt{A}, \ldots, \mathtt{Z}, \mathtt{a}, \ldots, \mathtt{z}\}^{3..20}$

Nichtleere Wörter aus lateinischen Buchstaben mit mindestens drei und höchstens 20 Buchstaben.

* $x \bmod m$: Rest bei der Division von x durch m.

Beispiel: $U = \{\mathtt{A}, \ldots, \mathtt{Z}, \mathtt{a}, \ldots, \mathtt{z}\}^{3..20}$

Nichtleere Wörter aus lateinischen Buchstaben mit mindestens drei und höchstens 20 Buchstaben.

$h\colon U \to [13]$ definiert durch

$$h(c_1 c_2 c_3 \cdots c_r) = \mathsf{num}(c_3) \bmod 13\,,$$

wobei

$$\begin{array}{lcl} \mathsf{num}(\mathtt{A}) = \mathsf{num}(\mathtt{a}) & = & 0 \\ \mathsf{num}(\mathtt{B}) = \mathsf{num}(\mathtt{b}) & = & 1 \\ \vdots \qquad\qquad \vdots & & \vdots \\ \mathsf{num}(\mathtt{Z}) = \mathsf{num}(\mathtt{z}) & = & 25 \end{array}$$

* $x \bmod m$: Rest bei der Division von x durch m.

In Tabelle:

In Tabelle:

x	$f(x)$	$\mathsf{num}(c_3)$	$h(x)$
Januar	31	13	0
Februar	28	1	1
Maerz	31	4	4
April	30	17	4
Mai	31	8	8
Juni	30	13	0
Juli	31	11	11
August	31	6	6
September	30	15	2
Oktober	31	19	6
November	30	21	8
Dezember	31	25	12

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

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)

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

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 $j = 0, 4, 6, 8$.

Definition 5.1.1

Eine Funktion $h\colon U \to [m]$ heißt eine **Hashfunktion**.

Definition 5.1.1

Eine Funktion $h\colon U \to [m]$ heißt eine **Hashfunktion**.

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

Definition 5.1.1

Eine Funktion $h\colon U \to [m]$ heißt eine **Hashfunktion**.

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

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

Definition 5.1.1

Eine Funktion $h\colon U \to [m]$ heißt eine **Hashfunktion**.

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

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

Definition 5.1.1

Eine Funktion $h\colon U \to [m]$ heißt eine **Hashfunktion**.

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

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

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

Definition 5.1.1

Eine Funktion $h\colon U \to [m]$ heißt eine **Hashfunktion**.

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

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

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

Beispiel: Auf $S_1 = \{\texttt{Februar}^{(1)}, \texttt{Maerz}^{(4)}, \texttt{Mai}^{(8)}, \texttt{Juni}^{(0)}, \texttt{Juli}^{(11)}, \texttt{Dezember}^{(12)}\}$ ist die Funktion h von oben injektiv,

Definition 5.1.1

Eine Funktion $h\colon U \to [m]$ heißt eine **Hashfunktion**.

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

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

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

Beispiel: Auf $S_1 = \{\texttt{Februar}^{(1)}, \texttt{Maerz}^{(4)}, \texttt{Mai}^{(8)}, \texttt{Juni}^{(0)}, \texttt{Juli}^{(11)}, \texttt{Dezember}^{(12)}\}$
ist die Funktion h von oben injektiv,
aber nicht auf $S_2 = \{\texttt{Januar}^{(0)}, \texttt{Mai}^{(8)}, \texttt{Juli}^{(11)}, \texttt{November}^{(8)}\}$.

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

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

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

lookup(x):

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

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

lookup(x):

- Berechne $j = h(x)$.
  (Dies sollte **effizient** möglich sein.)

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

Falls h perfekt für S:
Speichere x bzw. (x, r) in Tabellenplatz $\texttt{T}[h(x)]$, für $x \in S$.

lookup(x):

- Berechne $j = h(x)$.
  (Dies sollte **effizient** möglich sein.)
- Prüfe, ob in $\texttt{T}[j]$ Schlüssel x steht.

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

Falls h perfekt für S:
Speichere x bzw. (x, r) in Tabellenplatz $\texttt{T}[h(x)]$, für $x \in S$.

lookup(x):

- Berechne $j = h(x)$.
  (Dies sollte **effizient** möglich sein.)
- Prüfe, ob in $\texttt{T}[j]$ Schlüssel x steht.
  Falls ja: Inhalt (x, r) von $\texttt{T}[j]$ ausgeben, sonst „$x \notin S$“.

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

Falls h perfekt für S:
Speichere x bzw. (x, r) in Tabellenplatz $\texttt{T}[h(x)]$, für $x \in S$.

lookup(x):

- Berechne $j = h(x)$.
  (Dies sollte **effizient** möglich sein.)
- Prüfe, ob in $\texttt{T}[j]$ Schlüssel x steht.
  Falls ja: Inhalt (x, r) von $\texttt{T}[j]$ ausgeben, sonst „$x \notin S$“.

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

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

Falls h perfekt für S:
Speichere x bzw. (x, r) in Tabellenplatz $\texttt{T}[h(x)]$, für $x \in S$.

lookup(x):

- Berechne $j = h(x)$.
  (Dies sollte **effizient** möglich sein.)
- Prüfe, ob in $\texttt{T}[j]$ Schlüssel x steht.
  Falls ja: Inhalt (x, r) von $\texttt{T}[j]$ ausgeben, sonst „$x \notin S$“.

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

Leider sind Kollisionen „praktisch unvermeidlich“.

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

Falls h perfekt für S:
Speichere x bzw. (x, r) in Tabellenplatz $\texttt{T}[h(x)]$, für $x \in S$.

lookup(x):

- Berechne $j = h(x)$.
  (Dies sollte **effizient** möglich sein.)
- Prüfe, ob in $\texttt{T}[j]$ Schlüssel x steht.
  Falls ja: Inhalt (x, r) von $\texttt{T}[j]$ ausgeben, sonst „$x \notin S$“.

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

Leider sind Kollisionen „praktisch unvermeidlich“.

(Ausnahme: Speziell auf S abgestimmte Hashfunktion h,
fortgeschrittene Konstruktionen.)

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

* $\lfloor y \rfloor$ (bzw. $\lceil y \rceil$): größte (kleinste) ganze Zahl $\leq$ (bzw. $\geq$) y.

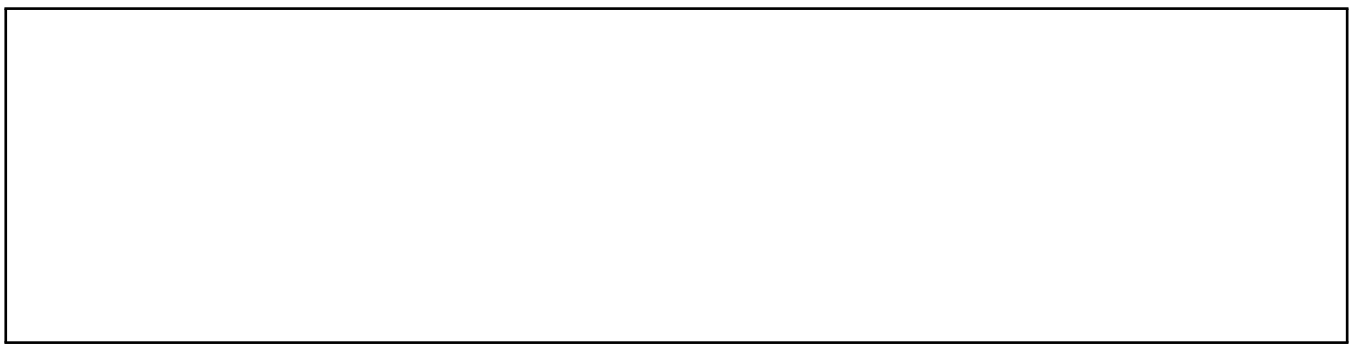
Fall: h „rein zufällig“, $|S| = n$, $S = \{x_1, \ldots, x_n\}$. D. h.:

Uniformitätsannahme (UF*):

* $\lfloor y \rfloor$ (bzw. $\lceil y \rceil$): größte (kleinste) ganze Zahl $\leq$ (bzw. $\geq$) y.

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

Uniformitätsannahme (UF*):



* $\lfloor y \rfloor$ (bzw. $\lceil y \rceil$): größte (kleinste) ganze Zahl $\leq$ (bzw. $\geq$) y.

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

Uniformitätsannahme (UF*):

Werte $h(x_1), \ldots, h(x_n)$ in $[m]^n$ sind „rein zufällig“:

* $\lfloor y \rfloor$ (bzw. $\lceil y \rceil$): größte (kleinste) ganze Zahl $\leq$ (bzw. $\geq$) y.

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

Uniformitätsannahme (UF*):

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

* $\lfloor y \rfloor$ (bzw. $\lceil y \rceil$): größte (kleinste) ganze Zahl $\leq$ (bzw. $\geq$) y.

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

Uniformitätsannahme (UF*):

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

(UF*) tritt z. B. (näherungsweise) ein, wenn

(i) U endlich ist

* $\lfloor y \rfloor$ (bzw. $\lceil y \rceil$): größte (kleinste) ganze Zahl $\leq$ (bzw. $\geq$) y.

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

Uniformitätsannahme (UF*):

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

(UF*) tritt z. B. (näherungsweise) ein, wenn

(i) U endlich ist und

(ii) h das Universum in gleich große Teile zerlegt, d. h.[*]

$$\left|h^{-1}(\{j\})\right| = \lfloor |U|/m \rfloor \quad \text{oder} \quad = \lceil |U|/m \rceil, \text{ für } j = 0, \ldots, m-1, \quad \text{und}$$

[*] $\lfloor y \rfloor$ (bzw. $\lceil y \rceil$): größte (kleinste) ganze Zahl $\leq$ (bzw. $\geq$) y.

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

Uniformitätsannahme (UF*):

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

(UF*) tritt z. B. (näherungsweise) ein, wenn

(i) U endlich ist und

(ii) h das Universum in gleich große Teile zerlegt, d. h.[*]

$$\left|h^{-1}(\{j\})\right| = \lfloor |U|/m \rfloor \quad \text{oder} \quad = \lceil |U|/m \rceil, \text{ für } j = 0, \ldots, m-1, \quad \text{und}$$

(iii) $S \subseteq U$ eine rein zufällige Menge der Größe n ist. (**(!) Sehr idealisierend!**)

[*] $\lfloor y \rfloor$ (bzw. $\lceil y \rceil$): größte (kleinste) ganze Zahl $\leq$ (bzw. $\geq$) y.

Mit (UF*):

Mit (UF*):

$$\mathbf{Pr}(\text{keine Kollision})$$

Mit (UF*):

$$\textbf{Pr}(\text{keine Kollision}) = \textbf{Pr}(h \text{ ist injektiv auf } \{x_1, \ldots, x_n\}) =$$

Mit (UF*):

$$\mathbf{Pr}(\text{keine Kollision}) = \mathbf{Pr}(h \text{ ist injektiv auf } \{x_1, \ldots, x_n\}) =$$

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

Mit (UF*):

$$\textbf{Pr}(\text{keine Kollision}) = \textbf{Pr}(h \text{ ist injektiv auf } \{x_1, \ldots, x_n\}) =$$
$$= \frac{m}{m} \cdot \frac{m-1}{m}$$

Mit (UF*):

$$\textbf{Pr}(\text{keine Kollision}) = \textbf{Pr}(h \text{ ist injektiv auf } \{x_1, \ldots, x_n\}) =$$
$$= \frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m}$$

Mit (UF*):

$$\textbf{Pr}(\text{keine Kollision}) = \textbf{Pr}(h \text{ ist injektiv auf } \{x_1, \ldots, x_n\}) =$$
$$= \frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdot \ldots \cdot \frac{m-n+1}{m} = \ldots$$

Mit (UF*):

$$\textbf{Pr}(\text{keine Kollision}) = \textbf{Pr}(h \text{ ist injektiv auf } \{x_1, \ldots, x_n\}) =$$
$$= \frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdot \ldots \cdot \frac{m-n+1}{m} = \ldots$$

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

Mit (UF*):

$$\textbf{Pr}(\text{keine Kollision}) = \textbf{Pr}(h \text{ ist injektiv auf } \{x_1, \ldots, x_n\}) =$$
$$= \frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdot \ldots \cdot \frac{m-n+1}{m} = \ldots$$

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

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

Mit (UF*):

$$\textbf{Pr}(\text{keine Kollision}) = \textbf{Pr}(h \text{ ist injektiv auf } \{x_1, \ldots, x_n\}) =$$
$$= \frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdot \ldots \cdot \frac{m-n+1}{m} = \ldots$$

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

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

Mit (UF*):

$$\textbf{Pr}(\text{keine Kollision}) = \textbf{Pr}(h \text{ ist injektiv auf } \{x_1, \ldots, x_n\}) =$$
$$= \frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdot \ldots \cdot \frac{m-n+1}{m} = \ldots$$

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

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

Mit (UF*):

$$\textbf{Pr}(\text{keine Kollision}) = \textbf{Pr}(h \text{ ist injektiv auf } \{x_1, \ldots, x_n\}) =$$
$$= \frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdot \ldots \cdot \frac{m-n+1}{m} = \ldots$$

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

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

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

Mit (UF*):

$$\textbf{Pr}(\text{keine Kollision}) = \textbf{Pr}(h \text{ ist injektiv auf } \{x_1, \ldots, x_n\}) =$$
$$= \frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdot \ldots \cdot \frac{m-n+1}{m} = \ldots$$

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

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

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

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

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

“Geburtstagsparadoxon”:

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

“Geburtstagsparadoxon”:

$x_1, \ldots, x_n$ sind Leute, $h(x_1), \ldots, h(x_n) \in [m]$ die Nummern ihrer (zufälligen) Geburtstage im Jahr, $m = 365$.

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

“Geburtstagsparadoxon”:

$x_1, \ldots, x_n$ sind Leute, $h(x_1), \ldots, h(x_n) \in [m]$ die Nummern ihrer (zufälligen) Geburtstage im Jahr, $m = 365$.

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

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

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

"Geburtstagsparadoxon":

$x_1, \ldots, x_n$ sind Leute, $h(x_1), \ldots, h(x_n) \in [m]$ die Nummern ihrer (zufälligen) Geburtstage im Jahr, $m = 365$.

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

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

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

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

Wunsch: Linearer Platzverbrauch, also

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

Wunsch: Linearer Platzverbrauch, also

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

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

Wunsch: Linearer Platzverbrauch, also

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

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

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

Wunsch: Linearer Platzverbrauch, also

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

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

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

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

Wunsch: Linearer Platzverbrauch, also

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

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

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

Winzig für nicht ganz kleine n!

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

Wunsch: Linearer Platzverbrauch, also

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

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

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

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

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

Wunsch: Linearer Platzverbrauch, also

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

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

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

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

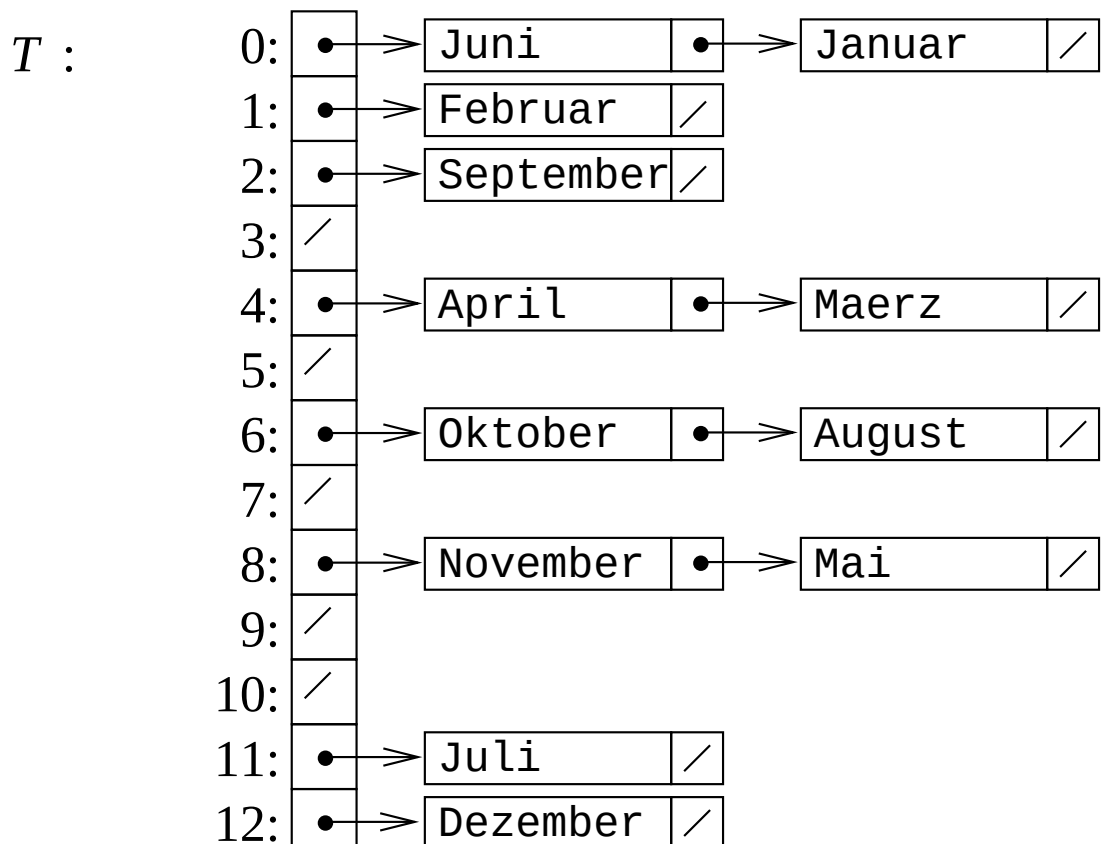
$\rightarrow$ **Kollisionsbehandlung** ist notwendig.

Vorlesungsvideo:

Hashing mit verketteten Listen

5.2 Hashing „mit verketteten Listen“

Beispiel:



5.2 Hashing „mit verketteten Listen“

engl.: **chained hashing**

5.2 Hashing „mit verketteten Listen“

engl.: **chained hashing**

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

5.2 Hashing „mit verketteten Listen“

engl.: **chained hashing**

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

In $\texttt{T}[0\,..\,m-1]$: **Zeiger** auf die Anfänge
von

5.2 Hashing „mit verketteten Listen“

engl.: **chained hashing**

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

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

5.2 Hashing „mit verketteten Listen“

engl.: **chained hashing**

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

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

(Oder auch: Zeiger auf „kleine“ Arrays/Vektoren mit variabler Länge.)

5.2 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** $L_0, \ldots, L_{m-1}$.

(Oder auch: Zeiger auf „kleine“ Arrays/Vektoren mit variabler Länge.)

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

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

5.2 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** $L_0, \ldots, L_{m-1}$.

(Oder auch: Zeiger auf „kleine“ Arrays/Vektoren mit variabler Länge.)

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

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

Bemerkung: Hier, als Unterstruktur, sind lineare Listen perfekt geeignet!

Wir implementieren die Wörterbuchoperationen: **Algorithmen(skizzen)**.

Wir implementieren die Wörterbuchoperationen: **Algorithmen(skizzen)**.

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

Wir implementieren die Wörterbuchoperationen: **Algorithmen(skizzen)**.

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

Wir implementieren die Wörterbuchoperationen: **Algorithmen(skizzen)**.

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

Kosten:

Wir implementieren die Wörterbuchoperationen: **Algorithmen(skizzen)**.

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

Kosten: $\Theta(m)$.

Wir implementieren die Wörterbuchoperationen: **Algorithmen(skizzen)**.

empty(m): Erzeugt Array `T[0..`$m-1$`]` mit m NULL-Zeigern/ NULL-Referenzen **und legt $h: U \to [m]$ 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 durch Durchlaufen der Liste L_j bei `T[`j`]`;

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

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

Kosten/Zeit: $O(1 +$ Zahl der Schlüsselvergleiche „$x = y$?“$)$

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

Kosten/Zeit: $O(1 +$ Zahl der Schlüsselvergleiche „$x = y$?“$)$

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

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

Kosten/Zeit: $O(1 +$ Zahl der Schlüsselvergleiche „$x = y$?“$)$

. . . 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 durch Durchlaufen der Liste L_j bei `T[`j`]`;
falls Eintrag (x, r) gefunden: **return** r;
// **erfolgreiche Suche**
sonst: **return** „*nicht vorhanden*“
// **erfolglose Suche**

Kosten/Zeit: $O(1 +$ Zahl der Schlüsselvergleiche „$x = y$?“$)$

. . . 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: 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: ersetze Daten bei x durch r;
// **erfolgreiche Suche**
sonst: füge (x, r) in Liste `T[`j`]` ein.
// **erfolglose Suche, Normalfall!**

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

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: ersetze Daten bei x durch r;
// **erfolgreiche Suche**
sonst: füge (x, r) in Liste `T[`j`]` ein.
// **erfolglose Suche, Normalfall!**

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: ersetze Daten bei x durch r;
// **erfolgreiche Suche**
sonst: füge (x, r) in Liste `T[`j`]` ein.
// **erfolglose Suche, Normalfall!**

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

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

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

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

delete(x): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei `T[`j`]`;
falls gefunden: lösche Eintrag $(x, f(x))$ aus L_j;
// **erfolgreiche Suche, Normalfall!**
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 ($\mathbf{UF}_2$)“:

Für je 2 Schlüssel $x \neq z$ in U gilt: $\mathbf{Pr}(h(x) = h(z)) \leq \frac{1}{m}$.

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 ($\mathbf{UF}_2$)“:

Für je 2 Schlüssel $x \neq z$ in U gilt: $\mathbf{Pr}(h(x) = h(z)) \leq \frac{1}{m}$.

Später: Wann ist diese Annahme erfüllt?

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 ($\mathbf{UF}_2$)“:

Für je 2 Schlüssel $x \neq z$ in U gilt: $\mathbf{Pr}(h(x) = h(z)) \leq \frac{1}{m}$.

Später: Wann ist diese Annahme erfüllt?
Wenn die Hashwerte rein zufällig sind, gilt: $\mathbf{Pr}(h(x) = h(z)) = \frac{1}{m}$.

Szenario:

Szenario: Menge $S = \{x_1, \ldots, x_n\}$ von n Schlüsseln in Struktur gespeichert.

Szenario: Menge $S = \{x_1, \ldots, x_n\}$ von n Schlüsseln in Struktur gespeichert.
Nun wird nach $y \in U$ gesucht.

Szenario: Menge $S = \{x_1, \ldots, x_n\}$ von n Schlüsseln in Struktur gespeichert. Nun wird nach $y \in U$ gesucht.

$\alpha = \frac{n}{m}$: der (aktuelle) **Auslastungsfaktor**.

Szenario: Menge $S = \{x_1, \ldots, x_n\}$ von n Schlüsseln in Struktur gespeichert. Nun wird nach $y \in U$ gesucht.

$\alpha = \frac{n}{m}$: der (aktuelle) **Auslastungsfaktor**.

Erfolglose Suche: $y \notin S$.

Szenario: Menge $S = \{x_1, \ldots, x_n\}$ von n Schlüsseln in Struktur gespeichert. Nun wird nach $y \in U$ gesucht.

$\alpha = \frac{n}{m}$: der (aktuelle) **Auslastungsfaktor**.

Erfolglose Suche: $y \notin S$.

Definiere **Zufallsgrößen/Zufallsvariable**, für alle $x, z \in U$:

Szenario: Menge $S = \{x_1, \ldots, x_n\}$ von n Schlüsseln in Struktur gespeichert. Nun wird nach $y \in U$ gesucht.

$\alpha = \frac{n}{m}$: der (aktuelle) **Auslastungsfaktor**.

Erfolglose Suche: $y \notin S$.

Definiere **Zufallsgrößen/Zufallsvariable**, für alle $x, z \in U$:

$$Y_{x,z} := \begin{cases} 1\ , & \text{falls } h(x) = h(z), \\ 0\ , & \text{falls } h(x) \neq h(z). \end{cases}$$

Szenario: Menge $S = \{x_1, \ldots, x_n\}$ von n Schlüsseln in Struktur gespeichert. Nun wird nach $y \in U$ gesucht.

$\alpha = \frac{n}{m}$: der (aktuelle) **Auslastungsfaktor**.

Erfolglose Suche: $y \notin S$.

Definiere **Zufallsgrößen/Zufallsvariable**, für alle $x, z \in U$:

$$Y_{x,z} := \begin{cases} 1\ , & \text{falls } h(x) = h(z), \\ 0\ , & \text{falls } h(x) \neq h(z). \end{cases}$$

Nach (UF_2):

Szenario: Menge $S = \{x_1, \ldots, x_n\}$ von n Schlüsseln in Struktur gespeichert.
Nun wird nach $y \in U$ gesucht.

$\alpha = \frac{n}{m}$: der (aktuelle) **Auslastungsfaktor**.

Erfolglose Suche: $y \notin S$.

Definiere **Zufallsgrößen/Zufallsvariable**, für alle $x, z \in U$:

$$Y_{x,z} := \begin{cases} 1\ , & \text{falls } h(x) = h(z), \\ 0\ , & \text{falls } h(x) \neq h(z). \end{cases}$$

Nach (UF_2): $\mathbf{E}(Y_{x,z}) = \mathbf{Pr}(h(x) = h(z)) \leq \frac{1}{m}$.

Szenario: Menge $S = \{x_1, \ldots, x_n\}$ von n Schlüsseln in Struktur gespeichert.
Nun wird nach $y \in U$ gesucht.

$\alpha = \frac{n}{m}$: der (aktuelle) **Auslastungsfaktor**.

Erfolglose Suche: $y \notin S$.

Definiere **Zufallsgrößen/Zufallsvariable**, für alle $x, z \in U$:

$$Y_{x,z} := \begin{cases} 1\ , & \text{falls } h(x) = h(z), \\ 0\ , & \text{falls } h(x) \neq h(z). \end{cases}$$

Nach (UF_2): $\mathbf{E}(Y_{x,z}) = \mathbf{Pr}(h(x) = h(z)) \leq \frac{1}{m}$.

Anzahl der Schlüsselvergleiche/Listenlänge $=$

Szenario: Menge $S = \{x_1, \ldots, x_n\}$ von n Schlüsseln in Struktur gespeichert.
Nun wird nach $y \in U$ gesucht.

$\alpha = \frac{n}{m}$: der (aktuelle) **Auslastungsfaktor**.

Erfolglose Suche: $y \notin S$.

Definiere **Zufallsgrößen/Zufallsvariable**, für alle $x, z \in U$:

$$Y_{x,z} := \begin{cases} 1\ , & \text{falls } h(x) = h(z), \\ 0\ , & \text{falls } h(x) \neq h(z). \end{cases}$$

Nach (UF_2): $\mathbf{E}(Y_{x,z}) = \mathbf{Pr}(h(x) = h(z)) \leq \frac{1}{m}$.

Anzahl der Schlüsselvergleiche/Listenlänge $=$

$$|B_{h(y)}|$$

Szenario: Menge $S = \{x_1, \ldots, x_n\}$ von n Schlüsseln in Struktur gespeichert. Nun wird nach $y \in U$ gesucht.

$\alpha = \frac{n}{m}$: der (aktuelle) **Auslastungsfaktor**.

Erfolglose Suche: $y \notin S$.

Definiere **Zufallsgrößen/Zufallsvariable**, für alle $x, z \in U$:

$$Y_{x,z} := \begin{cases} 1\ , & \text{falls } h(x) = h(z), \\ 0\ , & \text{falls } h(x) \neq h(z). \end{cases}$$

Nach (UF_2): $\mathbf{E}(Y_{x,z}) = \mathbf{Pr}(h(x) = h(z)) \leq \frac{1}{m}$.

Anzahl der Schlüsselvergleiche/Listenlänge $=$

$$|B_{h(y)}| = |\{x \in S \mid h(x) = h(y)\}|$$

Szenario: Menge $S = \{x_1, \ldots, x_n\}$ von n Schlüsseln in Struktur gespeichert.
Nun wird nach $y \in U$ gesucht.

$\alpha = \frac{n}{m}$: der (aktuelle) **Auslastungsfaktor**.

Erfolglose Suche: $y \notin S$.

Definiere **Zufallsgrößen/Zufallsvariable**, für alle $x, z \in U$:

$$Y_{x,z} := \begin{cases} 1\ , & \text{falls } h(x) = h(z), \\ 0\ , & \text{falls } h(x) \neq h(z). \end{cases}$$

Nach (UF_2): $\mathbf{E}(Y_{x,z}) = \mathbf{Pr}(h(x) = h(z)) \leq \frac{1}{m}$.

Anzahl der Schlüsselvergleiche/Listenlänge $=$

$$|B_{h(y)}| = |\{x \in S \mid h(x) = h(y)\}| = \sum\nolimits_{x \in S} Y_{x,y}.$$

Szenario: Menge $S = \{x_1, \ldots, x_n\}$ von n Schlüsseln in Struktur gespeichert. Nun wird nach $y \in U$ gesucht.

$\alpha = \frac{n}{m}$: der (aktuelle) **Auslastungsfaktor**.

Erfolglose Suche: $y \notin S$.

Definiere **Zufallsgrößen/Zufallsvariable**, für alle $x, z \in U$:

$$Y_{x,z} := \begin{cases} 1\ , & \text{falls } h(x) = h(z), \\ 0\ , & \text{falls } h(x) \neq h(z). \end{cases}$$

Nach (UF_2): $\mathbf{E}(Y_{x,z}) = \mathbf{Pr}(h(x) = h(z)) \leq \frac{1}{m}$.

Anzahl der Schlüsselvergleiche/Listenlänge $=$

$$|B_{h(y)}| = |\{x \in S \mid h(x) = h(y)\}| = \sum\nolimits_{x \in S} Y_{x,y}.$$

Eine Zufallsvariable!

Mit „Linearität des Erwartungswertes“:

Mit „Linearität des Erwartungswertes“:

$$\mathbf{E}(|B_{h(y)}|)$$

Mit „Linearität des Erwartungswertes“:

$$\mathbf{E}(|B_{h(y)}|) =$$

Mit „Linearität des Erwartungswertes“:

$$\mathbf{E}(|B_{h(y)}|) = \sum_{x \in S} \mathbf{E}(Y_{x,y})$$

Mit „Linearität des Erwartungswertes“:

$$\mathbf{E}(|B_{h(y)}|) = \sum_{x \in S} \mathbf{E}(Y_{x,y}) \overset{(\mathsf{UF}_2)}{\leq}$$

Mit „Linearität des Erwartungswertes“:

$$\mathbf{E}(|B_{h(y)}|) = \sum_{x \in S} \mathbf{E}(Y_{x,y}) \overset{(\mathsf{UF}_2)}{\leq} \sum_{x \in S} \frac{1}{m}$$

Mit „Linearität des Erwartungswertes“:

$$\mathbf{E}(|B_{h(y)}|) = \sum_{x \in S} \mathbf{E}(Y_{x,y}) \overset{(\mathsf{UF}_2)}{\leq} \sum_{x \in S} \frac{1}{m} = \frac{n}{m}$$

Mit „Linearität des Erwartungswertes“:

$$\mathbf{E}(|B_{h(y)}|) = \sum_{x \in S} \mathbf{E}(Y_{x,y}) \overset{(\mathsf{UF}_2)}{\leq} \sum_{x \in S} \frac{1}{m} = \frac{n}{m} = \alpha.$$

Mit „Linearität des Erwartungswertes“:

$$\mathbf{E}(|B_{h(y)}|) = \sum_{x \in S} \mathbf{E}(Y_{x,y}) \overset{(\mathsf{UF}_2)}{\leq} \sum_{x \in S} \frac{1}{m} = \frac{n}{m} = \alpha.$$

Satz 5.2.3

Bei Hashing mit verketteten Listen gilt unter der Annahme (UF_2):

Mit „Linearität des Erwartungswertes“:

$$\mathbf{E}(|B_{h(y)}|) = \sum_{x \in S} \mathbf{E}(Y_{x,y}) \overset{(\mathsf{UF}_2)}{\leq} \sum_{x \in S} \frac{1}{m} = \frac{n}{m} = \alpha.$$

Satz 5.2.3

Bei Hashing mit verketteten Listen gilt unter der Annahme (UF_2):

Die **erwartete Anzahl** von Schlüsselvergleichen bei **erfolgloser** Suche

Mit „Linearität des Erwartungswertes“:

$$\mathbf{E}(|B_{h(y)}|) = \sum_{x \in S} \mathbf{E}(Y_{x,y}) \overset{(\mathsf{UF}_2)}{\leq} \sum_{x \in S} \frac{1}{m} = \frac{n}{m} = \alpha.$$

Satz 5.2.3

Bei Hashing mit verketteten Listen gilt unter der Annahme (UF_2):

Die **erwartete Anzahl** von Schlüsselvergleichen bei **erfolgloser** Suche ist höchstens α, bei **erfolgreicher** Suche höchstens $1 + (n-1)/m < 1 + \alpha$.

Mit „Linearität des Erwartungswertes“:

$$\mathbf{E}(|B_{h(y)}|) = \sum_{x \in S} \mathbf{E}(Y_{x,y}) \overset{(\mathsf{UF}_2)}{\leq} \sum_{x \in S} \frac{1}{m} = \frac{n}{m} = \alpha.$$

Satz 5.2.3

Bei Hashing mit verketteten Listen gilt unter der Annahme (UF_2):

Die **erwartete Anzahl** von Schlüsselvergleichen bei **erfolgloser** Suche ist höchstens α, bei **erfolgreicher** Suche höchstens $1 + (n-1)/m < 1 + \alpha$.

(Analog; Summand $\mathbf{E}(Y_{y,y}) = 1$.)

Mit „Linearität des Erwartungswertes“:

$$\mathbf{E}(|B_{h(y)}|) = \sum_{x \in S} \mathbf{E}(Y_{x,y}) \overset{(\mathsf{UF}_2)}{\leq} \sum_{x \in S} \frac{1}{m} = \frac{n}{m} = \alpha.$$

Satz 5.2.3

Bei Hashing mit verketteten Listen gilt unter der Annahme (UF_2):

Die **erwartete Anzahl** von Schlüsselvergleichen bei **erfolgloser** Suche ist höchstens α, bei **erfolgreicher** Suche höchstens $1 + (n-1)/m < 1 + \alpha$.
(Analog; Summand $\mathbf{E}(Y_{y,y}) = 1$.)

Die **erwartete Zeit** für eine erfolglose/erfolgreiche Suche ist $O(1 + \alpha)$.

Mit „Linearität des Erwartungswertes“:

$$\mathbf{E}(|B_{h(y)}|) = \sum_{x \in S} \mathbf{E}(Y_{x,y}) \overset{(\mathsf{UF}_2)}{\leq} \sum_{x \in S} \frac{1}{m} = \frac{n}{m} = \alpha.$$

Satz 5.2.3

Bei Hashing mit verketteten Listen gilt unter der Annahme (UF_2):

Die **erwartete Anzahl** von Schlüsselvergleichen bei **erfolgloser** Suche ist höchstens α, bei **erfolgreicher** Suche höchstens $1 + (n-1)/m < 1 + \alpha$.
(Analog; Summand $\mathbf{E}(Y_{y,y}) = 1$.)

Die **erwartete Zeit** für eine erfolglose/erfolgreiche Suche ist $O(1 + \alpha)$.

Fazit: Wenn $\alpha \leq \alpha_1$, für α_1 konstant (z.B. $\alpha_1 = 2$), dann ist die **erwartete Zeit** für eine Suche $O(1)$.

Verdoppelungsstrategie

Verdoppelungsstrategie

. . . hilft, die Bedingung $\alpha \leq \alpha_1$ zu erzwingen, auch wenn anfangs die Schlüsselanzahl unbekannt ist.

(Analog zu Verdoppelungsstrategie bei Stacks, Kap. 2)

Verdoppelungsstrategie

. . . hilft, die Bedingung $\alpha \leq \alpha_1$ zu erzwingen, auch wenn anfangs die Schlüsselanzahl unbekannt ist.

(Analog zu Verdoppelungsstrategie bei Stacks, Kap. 2)

empty(m): Anfangs-Tabellengröße m

Verdoppelungsstrategie

. . . hilft, die Bedingung $\alpha \leq \alpha_1$ zu erzwingen, auch wenn anfangs die Schlüsselanzahl unbekannt ist.

(Analog zu Verdoppelungsstrategie bei Stacks, Kap. 2)

empty(m): Anfangs-Tabellengröße m oder
empty$()$: Anfangs-Tabellengröße m_0 (fester Wert)

Verdoppelungsstrategie

. . . hilft, die Bedingung $\alpha \leq \alpha_1$ zu erzwingen, auch wenn anfangs die Schlüsselanzahl unbekannt ist.

(Analog zu Verdoppelungsstrategie bei Stacks, Kap. 2)

empty(m): Anfangs-Tabellengröße m oder
empty$()$: Anfangs-Tabellengröße m_0 (fester Wert)

Variable `load` enthält aktuelle Anzahl von Schlüsseln.

Verdoppelungsstrategie

. . . hilft, die Bedingung $\alpha \leq \alpha_1$ zu erzwingen, auch wenn anfangs die Schlüsselanzahl unbekannt ist.

(Analog zu Verdoppelungsstrategie bei Stacks, Kap. 2)

empty(m): Anfangs-Tabellengröße m oder
empty$()$: Anfangs-Tabellengröße m_0 (fester Wert)

Variable `load` enthält aktuelle Anzahl von Schlüsseln.

empty(m): $\texttt{load} \leftarrow 0$,

Verdoppelungsstrategie

. . . hilft, die Bedingung $\alpha \leq \alpha_1$ zu erzwingen, auch wenn anfangs die Schlüsselanzahl unbekannt ist.

(Analog zu Verdoppelungsstrategie bei Stacks, Kap. 2)

empty(m): Anfangs-Tabellengröße m oder
empty$()$: Anfangs-Tabellengröße m_0 (fester Wert)

Variable `load` enthält aktuelle Anzahl von Schlüsseln.

empty(m):	$\texttt{load} \leftarrow 0$,
insert(x, r) mit x neu:	$\texttt{load} \leftarrow \texttt{load} + 1$,

Verdoppelungsstrategie

. . . hilft, die Bedingung $\alpha \leq \alpha_1$ zu erzwingen, auch wenn anfangs die Schlüsselanzahl unbekannt ist.

(Analog zu Verdoppelungsstrategie bei Stacks, Kap. 2)

empty(m): Anfangs-Tabellengröße m oder
empty$()$: Anfangs-Tabellengröße m_0 (fester Wert)

Variable `load` enthält aktuelle Anzahl von Schlüsseln.

empty(m):	$\texttt{load} \leftarrow 0$,
insert(x, r) mit x neu:	$\texttt{load} \leftarrow \texttt{load} + 1$,
delete(x) mit x vorhanden:	$\texttt{load} \leftarrow \texttt{load} - 1$.

Verdoppelungsstrategie (Fortsetzung)

Verdoppelungsstrategie (Fortsetzung)

Wenn nach *insert*(x, r) (mit x neu) der Wert `load` die Grenze $\alpha_1 m$ überschreitet:

Verdoppelungsstrategie (Fortsetzung)

Wenn nach *insert*(x, r) (mit x neu) der Wert `load` die Grenze $\alpha_1 m$ überschreitet:

- Erzeuge **neue Tabelle** $\mathtt{T}_{\text{neu}}$ der Größe $2m$.

Verdoppelungsstrategie (Fortsetzung)

Wenn nach *insert*(x, r) (mit x neu) der Wert `load` die Grenze $\alpha_1 m$ überschreitet:

- Erzeuge **neue Tabelle** $\mathtt{T}_{\text{neu}}$ der Größe $2m$.
- Erzeuge **neue Hashfunktion** $h_{\text{neu}} : U \to [2m]$.

Verdoppelungsstrategie (Fortsetzung)

Wenn nach *insert*(x, r) (mit x neu) der Wert `load` die Grenze $\alpha_1 m$ überschreitet:

- Erzeuge **neue Tabelle** $\mathtt{T}_{\text{neu}}$ der Größe $2m$.
- Erzeuge **neue Hashfunktion** $h_{\text{neu}} : U \to [2m]$.
- Übertrage Einträge aus den alten Listen
  in neue Listen zu Tabelle $\mathtt{T}_{\text{neu}}$, gemäß neuer Hashfunktion.

Verdoppelungsstrategie (Fortsetzung)

Wenn nach *insert*(x, r) (mit x neu) der Wert `load` die Grenze $\alpha_1 m$ überschreitet:

- Erzeuge **neue Tabelle** $\mathtt{T}_{\text{neu}}$ der Größe $2m$.
- Erzeuge **neue Hashfunktion** $h_{\text{neu}} : U \to [2m]$.
- Übertrage Einträge aus den alten Listen
  in neue Listen zu Tabelle $\mathtt{T}_{\text{neu}}$, gemäß neuer Hashfunktion.
  Dabei Einfügen **am Listenanfang ohne Suchen**.
  (Wissen: Alle Schlüssel sind verschieden!)

Verdoppelungsstrategie (Fortsetzung)

Wenn nach *insert*(x, r) (mit x neu) der Wert `load` die Grenze $\alpha_1 m$ überschreitet:

- Erzeuge **neue Tabelle** $\mathtt{T}_{\text{neu}}$ der Größe $2m$.
- Erzeuge **neue Hashfunktion** $h_{\text{neu}} : U \to [2m]$.
- Übertrage Einträge aus den alten Listen
  in neue Listen zu Tabelle $\mathtt{T}_{\text{neu}}$, gemäß neuer Hashfunktion.
  Dabei Einfügen **am Listenanfang ohne Suchen**.
  (Wissen: Alle Schlüssel sind verschieden!)
  Kosten: $O(m)$ im schlechtesten Fall.

Verdoppelungsstrategie (Fortsetzung)

Wenn nach *insert*(x, r) (mit x neu) der Wert `load` die Grenze $\alpha_1 m$ überschreitet:

- Erzeuge **neue Tabelle** $\mathtt{T}_{\text{neu}}$ der Größe $2m$.
- Erzeuge **neue Hashfunktion** $h_{\text{neu}} : U \to [2m]$.
- Übertrage Einträge aus den alten Listen
  in neue Listen zu Tabelle $\mathtt{T}_{\text{neu}}$, gemäß neuer Hashfunktion.
  Dabei Einfügen **am Listenanfang ohne Suchen**.
  (Wissen: Alle Schlüssel sind verschieden!)
  Kosten: $O(m)$ im schlechtesten Fall.
- Benenne $\mathtt{T}_{\text{neu}}$ in `T` und h_{neu} in h um.

Satz 5.2.4

Satz 5.2.4

Sei α_1 fest. Wenn die verwendeten Hashfunktionen (UF_2) erfüllen,

Satz 5.2.4

Sei α_1 fest. Wenn die verwendeten Hashfunktionen (UF_2) erfüllen, dann gilt für das skizzierte Verfahren für Hashing mit Listen mit Verdoppelungsstrategie:

Satz 5.2.4

Sei α_1 fest. Wenn die verwendeten Hashfunktionen (UF_2) erfüllen, dann gilt für das skizzierte Verfahren für Hashing mit Listen mit Verdoppelungsstrategie:

(a) Die erwartete Zeit für erfolglose bzw. erfolgreiche Suche ist $O(1 + \alpha_1)$.

Satz 5.2.4

Sei α_1 fest. Wenn die verwendeten Hashfunktionen (UF_2) erfüllen, dann gilt für das skizzierte Verfahren für Hashing mit Listen mit Verdoppelungsstrategie:

(a) Die erwartete Zeit für erfolglose bzw. erfolgreiche Suche ist $O(1 + \alpha_1)$.

(b) Die Gesamtkosten für sämtliche Verdopplungen

Satz 5.2.4

Sei α_1 fest. Wenn die verwendeten Hashfunktionen (UF_2) erfüllen, dann gilt für das skizzierte Verfahren für Hashing mit Listen mit Verdoppelungsstrategie:

(a) Die erwartete Zeit für erfolglose bzw. erfolgreiche Suche ist $O(1 + \alpha_1)$.

(b) Die Gesamtkosten für sämtliche Verdopplungen sind durch $O(\tilde{n})$ (schlechtester Fall) beschränkt;

Satz 5.2.4

Sei α_1 fest. Wenn die verwendeten Hashfunktionen (UF_2) erfüllen, dann gilt für das skizzierte Verfahren für Hashing mit Listen mit Verdoppelungsstrategie:

(a) Die erwartete Zeit für erfolglose bzw. erfolgreiche Suche ist $O(1 + \alpha_1)$.

(b) Die Gesamtkosten für sämtliche Verdopplungen sind durch $O(\tilde{n})$ (schlechtester Fall) beschränkt; dabei ist $\tilde{n}$ die maximale Anzahl der jemals gleichzeitig in der Datenstruktur vorhandenen Schlüssel (maximaler Wert in `load`).

Satz 5.2.4

Sei α_1 fest. Wenn die verwendeten Hashfunktionen (UF_2) erfüllen, dann gilt für das skizzierte Verfahren für Hashing mit Listen mit Verdoppelungsstrategie:

(a) Die erwartete Zeit für erfolglose bzw. erfolgreiche Suche ist $O(1 + \alpha_1)$.

(b) Die Gesamtkosten für sämtliche Verdopplungen sind durch $O(\tilde{n})$ (schlechtester Fall) beschränkt; dabei ist $\tilde{n}$ die maximale Anzahl der jemals gleichzeitig in der Datenstruktur vorhandenen Schlüssel (maximaler Wert in `load`).

(c) Der Gesamtplatzbedarf ist $O(\tilde{n})$.

Satz 5.2.4

Sei α_1 fest. Wenn die verwendeten Hashfunktionen (UF_2) erfüllen, dann gilt für das skizzierte Verfahren für Hashing mit Listen mit Verdoppelungsstrategie:

(a) Die erwartete Zeit für erfolglose bzw. erfolgreiche Suche ist $O(1 + \alpha_1)$.

(b) Die Gesamtkosten für sämtliche Verdopplungen sind durch $O(\tilde{n})$ (schlechtester Fall) beschränkt; dabei ist $\tilde{n}$ die maximale Anzahl der jemals gleichzeitig in der Datenstruktur vorhandenen Schlüssel (maximaler Wert in `load`).

(c) Der Gesamtplatzbedarf ist $O(\tilde{n})$.

(Zu (b): **Analyse** der Verdoppelungsstrategie wie bei Stacks.)

Vorlesungsvideo:

Hashfunktionen

5.3 Hashfunktionen

5.3 Hashfunktionen

5.3.1 Einfache Hashfunktionen

5.3 Hashfunktionen

5.3.1 Einfache Hashfunktionen

Kriterien für Qualität:

5.3 Hashfunktionen

5.3.1 Einfache Hashfunktionen

Kriterien für Qualität:

1) **Zeit für Auswertung** von $h(x)$

5.3 Hashfunktionen

5.3.1 Einfache Hashfunktionen

Kriterien für Qualität:

1) **Zeit für Auswertung** von $h(x)$ (*Schnell*!)

5.3 Hashfunktionen

5.3.1 Einfache Hashfunktionen

Kriterien für Qualität:

1) **Zeit für Auswertung** von $h(x)$ (*Schnell*!)
2) **Wahrscheinlichkeit** für **Kollisionen**

5.3 Hashfunktionen

5.3.1 Einfache Hashfunktionen

Kriterien für Qualität:

1) **Zeit für Auswertung** von $h(x)$ (*Schnell*!)
2) **Wahrscheinlichkeit** für **Kollisionen**
   (unter „realistischen Annahmen“ über Schlüsselmenge S).

5.3 Hashfunktionen

5.3.1 Einfache Hashfunktionen

Kriterien für Qualität:

1) **Zeit für Auswertung** von $h(x)$ (*Schnell*!)

2) **Wahrscheinlichkeit** für **Kollisionen**
(unter „realistischen Annahmen“ über Schlüsselmenge S).
(*Klein*, am besten $\leq c/m$ für c konstant.)

5.3 Hashfunktionen

5.3.1 Einfache Hashfunktionen

Kriterien für Qualität:

1) **Zeit für Auswertung** von $h(x)$ (*Schnell*!)

2) **Wahrscheinlichkeit** für **Kollisionen**
(unter „realistischen Annahmen“ über Schlüsselmenge S).
(*Klein*, am besten $\leq c/m$ für c konstant.)

Für 1) entscheidend: **Schlüsselformat**.

1. Fall: Schlüssel sind natürliche Zahlen: $U \subseteq \mathbb{N}$.

1. Fall: Schlüssel sind natürliche Zahlen: $U \subseteq \mathbb{N}$.

Divisionsrestmethode: $\boxed{h(x) = x \bmod m}$

1. Fall: Schlüssel sind natürliche Zahlen: $U \subseteq \mathbb{N}$.

Divisionsrestmethode: $\boxed{h(x) = x \bmod m}$

Sehr einfach, recht effizient.

1. Fall: Schlüssel sind natürliche Zahlen: $U \subseteq \mathbb{N}$.

Divisionsrestmethode: $\boxed{h(x) = x \bmod m}$

Sehr einfach, recht effizient.

Nachteile:

- Unflexibel: Nur eine feste Funktion für eine Tabellengröße.

1. Fall: Schlüssel sind natürliche Zahlen: $U \subseteq \mathbb{N}$.

Divisionsrestmethode: $\boxed{h(x) = x \bmod m}$

Sehr einfach, recht effizient.

Nachteile:

- Unflexibel: Nur eine feste Funktion für eine Tabellengröße.
- Bei strukturierten Schlüsselmengen (z. B. aus Strings übersetzte Schlüssel) deutlich erhöhtes Kollisionsrisiko.

1. Fall: Schlüssel sind natürliche Zahlen: $U \subseteq \mathbb{N}$.

Divisionsrestmethode: $\boxed{h(x) = x \bmod m}$

Sehr einfach, recht effizient.

Nachteile:

- Unflexibel: Nur eine feste Funktion für eine Tabellengröße.
- Bei strukturierten Schlüsselmengen (z. B. aus Strings übersetzte Schlüssel) deutlich erhöhtes Kollisionsrisiko.

Für Produktionssysteme darf diese Methode nicht benutzt werden!

Multiplikationsmethode:

Multiplikationsmethode: $U \subseteq \mathbb{N}$

Multiplikationsmethode: $U \subseteq \mathbb{N}$. *Idee:* Für $0 < \vartheta < 1$ (idealisiert: ϑ **irrational**)

Multiplikationsmethode: $U \subseteq \mathbb{N}$. *Idee:* Für $0 < \vartheta < 1$ (idealisiert: ϑ **irrational**)

$h(x) := \lfloor ((\vartheta \cdot x) \bmod 1) \cdot m \rfloor$.

Multiplikationsmethode: $U \subseteq \mathbb{N}$. *Idee:* Für $0 < \vartheta < 1$ (idealisiert: ϑ **irrational**)

$h(x) := \lfloor ((\vartheta \cdot x) \bmod 1) \cdot m \rfloor$. (Dabei: $(\vartheta \cdot x) \bmod 1 = \vartheta \cdot x - \lfloor \vartheta \cdot x \rfloor \in [0, 1)$.)

Multiplikationsmethode: $U \subseteq \mathbb{N}$. *Idee:* Für $0 < \vartheta < 1$ (idealisiert: ϑ **irrational**)

$h(x) := \lfloor ((\vartheta \cdot x) \bmod 1) \cdot m \rfloor$. (Dabei: $(\vartheta \cdot x) \bmod 1 = \vartheta \cdot x - \lfloor \vartheta \cdot x \rfloor \in [0, 1)$.)

(Gebrochener Anteil von ϑx wird auf Wertebereich $[0, m)$ „aufgeblasen“, dann auf ganze Zahl abgerundet. Für $\theta = \frac{1}{2}(\sqrt{5} - 1) \approx 1{,}618$ weiß man: $x, x + 1, \ldots, x + n - 1$ werden gut verteilt.)

Multiplikationsmethode: $U \subseteq \mathbb{N}$. *Idee:* Für $0 < \vartheta < 1$ (idealisiert: ϑ **irrational**)

$h(x) := \lfloor ((\vartheta \cdot x) \bmod 1) \cdot m \rfloor$. (Dabei: $(\vartheta \cdot x) \bmod 1 = \vartheta \cdot x - \lfloor \vartheta \cdot x \rfloor \in [0, 1)$.)

(Gebrochener Anteil von ϑx wird auf Wertebereich $[0, m)$ „aufgeblasen“, dann auf ganze Zahl abgerundet. Für $\theta = \frac{1}{2}(\sqrt{5} - 1) \approx 1{,}618$ weiß man: $x, x + 1, \ldots, x + n - 1$ werden gut verteilt.)

Diskrete Multiplikationsmethode:

Multiplikationsmethode: $U \subseteq \mathbb{N}$. *Idee:* Für $0 < \vartheta < 1$ (idealisiert: ϑ **irrational**)

$h(x) := \lfloor ((\vartheta \cdot x) \bmod 1) \cdot m \rfloor$. (Dabei: $(\vartheta \cdot x) \bmod 1 = \vartheta \cdot x - \lfloor \vartheta \cdot x \rfloor \in [0, 1)$.)

(Gebrochener Anteil von ϑx wird auf Wertebereich $[0, m)$ „aufgeblasen“, dann auf ganze Zahl abgerundet. Für $\theta = \frac{1}{2}(\sqrt{5} - 1) \approx 1{,}618$ weiß man: $x, x + 1, \ldots, x + n - 1$ werden gut verteilt.)

Diskrete Multiplikationsmethode: $U = [2^k]$ und $m = 2^l$.

Multiplikationsmethode: $U \subseteq \mathbb{N}$. *Idee:* Für $0 < \vartheta < 1$ (idealisiert: ϑ **irrational**)

$h(x) := \lfloor ((\vartheta \cdot x) \bmod 1) \cdot m \rfloor$. (Dabei: $(\vartheta \cdot x) \bmod 1 = \vartheta \cdot x - \lfloor \vartheta \cdot x \rfloor \in [0, 1)$.)

(Gebrochener Anteil von ϑx wird auf Wertebereich $[0, m)$ „aufgeblasen“, dann auf ganze Zahl abgerundet. Für $\theta = \frac{1}{2}(\sqrt{5} - 1) \approx 1{,}618$ weiß man: $x, x + 1, \ldots, x + n - 1$ werden gut verteilt.)

Diskrete Multiplikationsmethode: $U = [2^k]$ und $m = 2^l$.

$$h_a(x) = (ax \bmod 2^k) \operatorname{div} 2^{k-l}.$$

mit $a \in [2^k]$ ungerade und z. B. $a/2^k \approx \frac{1}{2}\left(\sqrt{5} - 1\right) = 0{,}618\ldots$.

Multiplikationsmethode: $U \subseteq \mathbb{N}$. *Idee:* Für $0 < \vartheta < 1$ (idealisiert: ϑ **irrational**)

$h(x) := \lfloor ((\vartheta \cdot x) \bmod 1) \cdot m \rfloor$. (Dabei: $(\vartheta \cdot x) \bmod 1 = \vartheta \cdot x - \lfloor \vartheta \cdot x \rfloor \in [0, 1)$.)

(Gebrochener Anteil von ϑx wird auf Wertebereich $[0, m)$ „aufgeblasen“, dann auf ganze Zahl abgerundet. Für $\theta = \frac{1}{2}(\sqrt{5} - 1) \approx 1{,}618$ weiß man: $x, x + 1, \ldots, x + n - 1$ werden gut verteilt.)

Diskrete Multiplikationsmethode: $U = [2^k]$ und $m = 2^l$.

$$h_a(x) = (ax \bmod 2^k) \operatorname{div} 2^{k-l}.$$

mit $a \in [2^k]$ ungerade und z. B. $a/2^k \approx \frac{1}{2}\left(\sqrt{5} - 1\right) = 0{,}618\ldots$.

Oder: $a \in [2^k]$ ungerade, **zufällig**.

Multiplikationsmethode: $U \subseteq \mathbb{N}$. *Idee:* Für $0 < \vartheta < 1$ (idealisiert: ϑ **irrational**)

$h(x) := \lfloor ((\vartheta \cdot x) \bmod 1) \cdot m \rfloor$. (Dabei: $(\vartheta \cdot x) \bmod 1 = \vartheta \cdot x - \lfloor \vartheta \cdot x \rfloor \in [0, 1)$.)

(Gebrochener Anteil von ϑx wird auf Wertebereich $[0, m)$ „aufgeblasen“, dann auf ganze Zahl abgerundet. Für $\theta = \frac{1}{2}(\sqrt{5} - 1) \approx 1{,}618$ weiß man: $x, x + 1, \ldots, x + n - 1$ werden gut verteilt.)

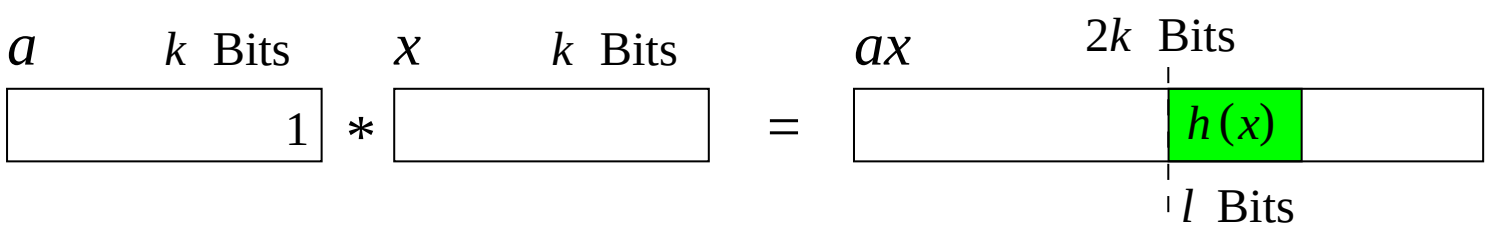
Diskrete Multiplikationsmethode: $U = [2^k]$ und $m = 2^l$.

$$h_a(x) = (ax \bmod 2^k) \operatorname{div} 2^{k-l}.$$

mit $a \in [2^k]$ ungerade und z. B. $a/2^k \approx \frac{1}{2}\left(\sqrt{5} - 1\right) = 0{,}618\ldots$.

Oder: $a \in [2^k]$ ungerade, **zufällig**.

Sehr effiziente Auswertung ohne Division:



Multiplikationsmethode: $U \subseteq \mathbb{N}$. *Idee:* Für $0 < \vartheta < 1$ (idealisiert: ϑ **irrational**)

$h(x) := \lfloor ((\vartheta \cdot x) \bmod 1) \cdot m \rfloor$. (Dabei: $(\vartheta \cdot x) \bmod 1 = \vartheta \cdot x - \lfloor \vartheta \cdot x \rfloor \in [0, 1)$.)

(Gebrochener Anteil von ϑx wird auf Wertebereich $[0, m)$ „aufgeblasen“, dann auf ganze Zahl abgerundet. Für $\theta = \frac{1}{2}(\sqrt{5} - 1) \approx 1{,}618$ weiß man: $x, x + 1, \ldots, x + n - 1$ werden gut verteilt.)

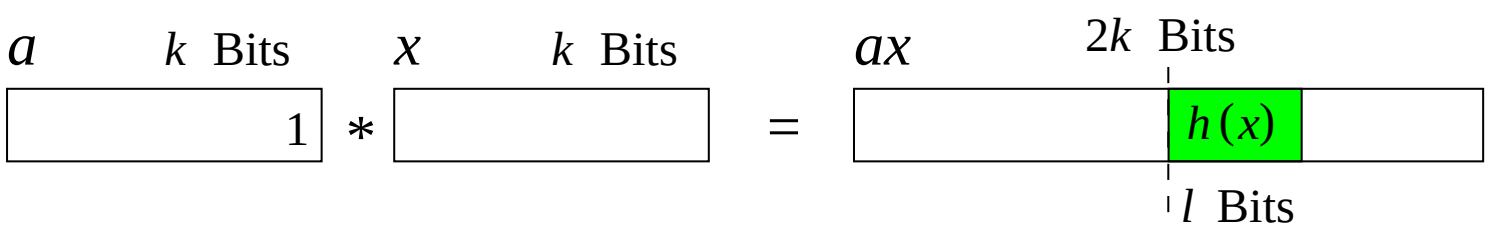
Diskrete Multiplikationsmethode: $U = [2^k]$ und $m = 2^l$.

$$h_a(x) = (ax \bmod 2^k) \operatorname{div} 2^{k-l}.$$

mit $a \in [2^k]$ ungerade und z. B. $a/2^k \approx \frac{1}{2}\left(\sqrt{5} - 1\right) = 0{,}618\ldots$.

Oder: $a \in [2^k]$ ungerade, **zufällig**.

Sehr effiziente Auswertung ohne Division:



(Wenn $k =$ Wortbreite w: Multiplikation + Shift.)

2. Fall: Schlüssel sind Strings/Wörter: $U \subseteq \mathsf{Seq}(\Sigma)$

2. Fall: Schlüssel sind Strings/Wörter: $U \subseteq \text{Seq}(\Sigma)$

Σ: „**Alphabet**“ – endliche nichtleere Menge.

2. Fall: Schlüssel sind Strings/Wörter: $U \subseteq \text{Seq}(\Sigma)$

Σ: **„Alphabet“** – endliche nichtleere Menge.

$\mathbf{Seq(\Sigma)}$: Menge aller endlichen Folgen von Elementen von Σ: „Wörter über Σ“.

2. Fall: Schlüssel sind Strings/Wörter: $U \subseteq \text{Seq}(\Sigma)$

Σ: **„Alphabet“** – endliche nichtleere Menge.

$\mathbf{Seq(\Sigma)}$: Menge aller endlichen Folgen von Elementen von Σ: „Wörter über Σ“.

Beispiel: $\Sigma =$ ISO-8859-1-Alphabet $\widehat{=}$ $\{0,1\}^8$ $\widehat{=}$ $\{0,1,\ldots,255\}$.

2. Fall: Schlüssel sind Strings/Wörter: $U \subseteq \text{Seq}(\Sigma)$

Σ: „**Alphabet**“ – endliche nichtleere Menge.

$\mathbf{Seq(\Sigma)}$: Menge aller endlichen Folgen von Elementen von Σ: „Wörter über Σ“.

Beispiel: $\Sigma =$ ISO-8859-1-Alphabet $\widehat{=}$ $\{0,1\}^8$ $\widehat{=}$ $\{0,1,\ldots,255\}$.

(Siehe `https://de.wikipedia.org/wiki/ISO_8859-1#ISO/IEC_8859-1`.)

2. Fall: Schlüssel sind Strings/Wörter: $U \subseteq \text{Seq}(\Sigma)$

Σ: **„Alphabet“** – endliche nichtleere Menge.

$\mathbf{Seq(\Sigma)}$: Menge aller endlichen Folgen von Elementen von Σ: „Wörter über Σ“.

Beispiel: $\Sigma =$ ISO-8859-1-Alphabet $\widehat{=}$ $\{0,1\}^8$ $\widehat{=}$ $\{0,1,\ldots,255\}$.

(Siehe `https://de.wikipedia.org/wiki/ISO_8859-1#ISO/IEC_8859-1`.)

Schlüssel: $x = (c_1, \ldots, c_r) = c_1 \ldots c_r$ mit $c_1, \ldots, c_r \in \Sigma$.

2. Fall: Schlüssel sind Strings/Wörter: $U \subseteq \text{Seq}(\Sigma)$

Σ: **„Alphabet“** – endliche nichtleere Menge.

$\mathbf{Seq(\Sigma)}$: Menge aller endlichen Folgen von Elementen von Σ: „Wörter über Σ“.

Beispiel: $\Sigma =$ ISO-8859-1-Alphabet $\widehat{=}$ $\{0,1\}^8$ $\widehat{=}$ $\{0,1,\ldots,255\}$.

(Siehe `https://de.wikipedia.org/wiki/ISO_8859-1#ISO/IEC_8859-1`.)

Schlüssel: $x = (c_1, \ldots, c_r) = c_1 \ldots c_r$ mit $c_1, \ldots, c_r \in \Sigma$.

Möglich, aber nicht zu empfehlen:

2. Fall: Schlüssel sind Strings/Wörter: $U \subseteq \text{Seq}(\Sigma)$

Σ: **„Alphabet“** – endliche nichtleere Menge.

$\mathbf{Seq(\Sigma)}$: Menge aller endlichen Folgen von Elementen von Σ: „Wörter über Σ“.

Beispiel: $\Sigma =$ ISO-8859-1-Alphabet $\widehat{=}$ $\{0,1\}^8$ $\widehat{=}$ $\{0,1,\ldots,255\}$.

(Siehe `https://de.wikipedia.org/wiki/ISO_8859-1#ISO/IEC_8859-1`.)

Schlüssel: $x = (c_1, \ldots, c_r) = c_1 \ldots c_r$ mit $c_1, \ldots, c_r \in \Sigma$.

Möglich, aber nicht zu empfehlen:
Transformation von Schlüssel x in Zahl $\hat{x} = n(x)$ als neuen Schlüssel,

2. Fall: Schlüssel sind Strings/Wörter: $U \subseteq \text{Seq}(\Sigma)$

Σ: **„Alphabet“** – endliche nichtleere Menge.

$\mathbf{Seq(\Sigma)}$: Menge aller endlichen Folgen von Elementen von Σ: „Wörter über Σ“.

Beispiel: $\Sigma =$ ISO-8859-1-Alphabet $\widehat{=}$ $\{0,1\}^8$ $\widehat{=}$ $\{0,1,\ldots,255\}$.

(Siehe `https://de.wikipedia.org/wiki/ISO_8859-1#ISO/IEC_8859-1`.)

Schlüssel: $x = (c_1, \ldots, c_r) = c_1 \ldots c_r$ mit $c_1, \ldots, c_r \in \Sigma$.

Möglich, aber nicht zu empfehlen:
Transformation von Schlüssel x in Zahl $\hat{x} = n(x)$ als neuen Schlüssel, dann: $x \mapsto h(\hat{x})$ für Hashfunktion h auf Zahlen.

2. Fall: Schlüssel sind Strings/Wörter: $U \subseteq \text{Seq}(\Sigma)$

Σ: **„Alphabet“** – endliche nichtleere Menge.

$\mathbf{Seq(\Sigma)}$: Menge aller endlichen Folgen von Elementen von Σ: „Wörter über Σ“.

Beispiel: $\Sigma =$ ISO-8859-1-Alphabet $\widehat{=} \{0,1\}^8 \widehat{=} \{0,1,\ldots,255\}$.

(Siehe `https://de.wikipedia.org/wiki/ISO_8859-1#ISO/IEC_8859-1`.)

Schlüssel: $x = (c_1, \ldots, c_r) = c_1 \ldots c_r$ mit $c_1, \ldots, c_r \in \Sigma$.

Möglich, aber nicht zu empfehlen:
Transformation von Schlüssel x in Zahl $\hat{x} = n(x)$ als neuen Schlüssel, dann: $x \mapsto h(\hat{x})$ für Hashfunktion h auf Zahlen.

- Kollisionen $n(x) = n(y)$ nie auflösbar!

2. Fall: Schlüssel sind Strings/Wörter: $U \subseteq \mathrm{Seq}(\Sigma)$

Σ: **„Alphabet“** – endliche nichtleere Menge.

$\mathbf{Seq(\Sigma)}$: Menge aller endlichen Folgen von Elementen von Σ: „Wörter über Σ“.

Beispiel: $\Sigma =$ ISO-8859-1-Alphabet $\widehat{=}$ $\{0,1\}^8$ $\widehat{=}$ $\{0,1,\ldots,255\}$.

(Siehe `https://de.wikipedia.org/wiki/ISO_8859-1#ISO/IEC_8859-1`.)

Schlüssel: $x = (c_1,\ldots,c_r) = c_1\ldots c_r$ mit $c_1,\ldots,c_r \in \Sigma$.

Möglich, aber nicht zu empfehlen:
Transformation von Schlüssel x in Zahl $\hat{x} = n(x)$ als neuen Schlüssel, dann: $x \mapsto h(\hat{x})$ für Hashfunktion h auf Zahlen.

- Kollisionen $n(x) = n(y)$ nie auflösbar!
- Selbst wenn n injektiv: Ineffiziente Auswertung von $h(\hat{x})$.

2. Fall: Schlüssel sind Strings/Wörter: $U \subseteq \mathrm{Seq}(\Sigma)$

Σ: **„Alphabet“** – endliche nichtleere Menge.

$\mathbf{Seq(\Sigma)}$: Menge aller endlichen Folgen von Elementen von Σ: „Wörter über Σ“.

Beispiel: $\Sigma =$ ISO-8859-1-Alphabet $\widehat{=}$ $\{0,1\}^8$ $\widehat{=}$ $\{0,1,\ldots,255\}$.

(Siehe `https://de.wikipedia.org/wiki/ISO_8859-1#ISO/IEC_8859-1`.)

Schlüssel: $x = (c_1,\ldots,c_r) = c_1\ldots c_r$ mit $c_1,\ldots,c_r \in \Sigma$.

Möglich, aber nicht zu empfehlen:
Transformation von Schlüssel x in Zahl $\hat{x} = n(x)$ als neuen Schlüssel, dann: $x \mapsto h(\hat{x})$ für Hashfunktion h auf Zahlen.

- Kollisionen $n(x) = n(y)$ nie auflösbar!
- Selbst wenn n injektiv: Ineffiziente Auswertung von $h(\hat{x})$.

Besser: An Schlüsselformat $x = (c_1,\ldots,c_r)$ angepasste Hashfunktionen.

Lineare Funktionen über Körper $\mathbb{Z}_p = [p] = \{0, \ldots, p-1\}$:

Lineare Funktionen über Körper $\mathbb{Z}_p = [p] = \{0, \ldots, p-1\}$:

$\Sigma \subseteq [p]$ für p Primzahl, $m = p$. (Beispiel: ISO-8859-1 $\widehat{=}\ [256] \subseteq [p] = [1009]$.)

Lineare Funktionen über Körper $\mathbb{Z}_p = [p] = \{0, \ldots, p-1\}$:

$\Sigma \subseteq [p]$ für p Primzahl, $m = p$. (Beispiel: ISO-8859-1 $\widehat{=}\ [256] \subseteq [p] = [1009]$.)

Wähle Koeffizienten $a_1, \ldots, a_r \in [p]$.

Lineare Funktionen über Körper $\mathbb{Z}_p = [p] = \{0, \ldots, p-1\}$:

$\Sigma \subseteq [p]$ für p Primzahl, $m = p$. (Beispiel: ISO-8859-1 $\widehat{=}\ [256] \subseteq [p] = [1009]$.)

Wähle Koeffizienten $a_1, \ldots, a_r \in [p]$.

$$h(c_1 \cdots c_r) = \left(\sum_{i=1}^{r} a_i \cdot c_i \right) \bmod p.$$

Lineare Funktionen über Körper $\mathbb{Z}_p = [p] = \{0, \ldots, p-1\}$:

$\Sigma \subseteq [p]$ für p Primzahl, $m = p$. (Beispiel: ISO-8859-1 $\widehat{=}\ [256] \subseteq [p] = [1009]$.)

Wähle Koeffizienten $a_1, \ldots, a_r \in [p]$.

$$h(c_1 \cdots c_r) = \left(\sum_{i=1}^{r} a_i \cdot c_i \right) \bmod p.$$

Vorteile:

- Rechnung nur mit Zahlen $< p^2 = m^2$

Lineare Funktionen über Körper $\mathbb{Z}_p = [p] = \{0, \ldots, p-1\}$:

$\Sigma \subseteq [p]$ für p Primzahl, $m = p$. (Beispiel: ISO-8859-1 $\widehat{=}\ [256] \subseteq [p] = [1009]$.)

Wähle Koeffizienten $a_1, \ldots, a_r \in [p]$.

$$h(c_1 \cdots c_r) = \left(\sum_{i=1}^{r} a_i \cdot c_i \right) \bmod p.$$

Vorteile:

- Rechnung nur mit Zahlen $< p^2 = m^2$ . . . (wenn man nach jeder Multiplikation/Addition Rest modulo p bildet);

Lineare Funktionen über Körper $\mathbb{Z}_p = [p] = \{0, \ldots, p-1\}$:

$\Sigma \subseteq [p]$ für p Primzahl, $m = p$. (Beispiel: ISO-8859-1 $\widehat{=}\ [256] \subseteq [p] = [1009]$.)

Wähle Koeffizienten $a_1, \ldots, a_r \in [p]$.

$$h(c_1 \cdots c_r) = \left(\sum_{i=1}^{r} a_i \cdot c_i \right) \bmod p.$$

Vorteile:

- Rechnung nur mit Zahlen $< p^2 = m^2$ . . . (wenn man nach jeder Multiplikation/Addition Rest modulo p bildet);
- sehr effizient;

Lineare Funktionen über Körper $\mathbb{Z}_p = [p] = \{0, \ldots, p-1\}$:

$\Sigma \subseteq [p]$ für p Primzahl, $m = p$. (Beispiel: ISO-8859-1 $\widehat{=}$ $[256] \subseteq [p] = [1009]$.)

Wähle Koeffizienten $a_1, \ldots, a_r \in [p]$.

$$h(c_1 \cdots c_r) = \left(\sum_{i=1}^{r} a_i \cdot c_i \right) \bmod p.$$

Vorteile:

- Rechnung nur mit Zahlen $< p^2 = m^2$ . . . (wenn man nach jeder Multiplikation/Addition Rest modulo p bildet);
- sehr effizient;
- theoretisch nachweisbares gutes Verhalten, wenn $a_1, \ldots, a_r$ aus $[p]$ **zufällig** gewählt werden.

5.3.2 Anspruchsvollere Hashfunktionen

Qualitätskriterien:

5.3.2 Anspruchsvollere Hashfunktionen

Qualitätskriterien:

1) Zeit für **Auswertung** von $h(x)$

5.3.2 Anspruchsvollere Hashfunktionen

Qualitätskriterien:

1) Zeit für **Auswertung** von $h(x)$ (*klein*!)

5.3.2 Anspruchsvollere Hashfunktionen

Qualitätskriterien:

1) Zeit für **Auswertung** von $h(x)$ (*klein*!)
2) geringe **Wahrscheinlichkeit** für **Kollisionen**

5.3.2 Anspruchsvollere Hashfunktionen

Qualitätskriterien:

1) Zeit für **Auswertung** von $h(x)$ (*klein*!)
2) geringe **Wahrscheinlichkeit** für **Kollisionen**
   ohne Annahmen über die Schlüsselmenge S

5.3.2 Anspruchsvollere Hashfunktionen

Qualitätskriterien:

1) Zeit für **Auswertung** von $h(x)$ (*klein*!)
2) geringe **Wahrscheinlichkeit** für **Kollisionen**
   ohne Annahmen über die Schlüsselmenge S (ähnlich der zufälligen Situation!)

5.3.2 Anspruchsvollere Hashfunktionen

Qualitätskriterien:

1) Zeit für **Auswertung** von $h(x)$ (*klein*!)
2) geringe **Wahrscheinlichkeit** für **Kollisionen**
   ohne Annahmen über die Schlüsselmenge S (ähnlich der zufälligen Situation!)
3) Effizientes **Generieren** einer neuen Hashfunktion

5.3.2 Anspruchsvollere Hashfunktionen

Qualitätskriterien:

1) Zeit für **Auswertung** von $h(x)$ (*klein*!)
2) geringe **Wahrscheinlichkeit** für **Kollisionen**
   ohne Annahmen über die Schlüsselmenge S (ähnlich der zufälligen Situation!)
3) Effizientes **Generieren** einer neuen Hashfunktion
   (benötigt z. B. bei der Verdoppelungsstrategie)

5.3.2 Anspruchsvollere Hashfunktionen

Qualitätskriterien:

1) Zeit für **Auswertung** von $h(x)$ (*klein*!)
2) geringe **Wahrscheinlichkeit** für **Kollisionen**
   ohne Annahmen über die Schlüsselmenge S (ähnlich der zufälligen Situation!)
3) Effizientes **Generieren** einer neuen Hashfunktion
   (benötigt z. B. bei der Verdoppelungsstrategie)
4) **Platzbedarf** für die Speicherung von h (Programmtext, Parameter)

5.3.2 Anspruchsvollere Hashfunktionen

Qualitätskriterien:

1) Zeit für **Auswertung** von $h(x)$ (*klein*!)
2) geringe **Wahrscheinlichkeit** für **Kollisionen**
   ohne Annahmen über die Schlüsselmenge S (ähnlich der zufälligen Situation!)
3) Effizientes **Generieren** einer neuen Hashfunktion
   (benötigt z. B. bei der Verdoppelungsstrategie)
4) **Platzbedarf** für die Speicherung von h (Programmtext, Parameter)
   (*gering* gegenüber n!)

5.3.2 Anspruchsvollere Hashfunktionen

Qualitätskriterien:

1) Zeit für **Auswertung** von $h(x)$ (*klein*!)
2) geringe **Wahrscheinlichkeit** für **Kollisionen**
   ohne Annahmen über die Schlüsselmenge S (ähnlich der zufälligen Situation!)
3) Effizientes **Generieren** einer neuen Hashfunktion
   (benötigt z. B. bei der Verdoppelungsstrategie)
4) **Platzbedarf** für die Speicherung von h (Programmtext, Parameter)
   (*gering* gegenüber n!)

Universelles Hashing (Idee)

5.3.2 Anspruchsvollere Hashfunktionen

Qualitätskriterien:

1) Zeit für **Auswertung** von $h(x)$ (*klein*!)
2) geringe **Wahrscheinlichkeit** für **Kollisionen**
   ohne Annahmen über die Schlüsselmenge S (ähnlich der zufälligen Situation!)
3) Effizientes **Generieren** einer neuen Hashfunktion
   (benötigt z. B. bei der Verdoppelungsstrategie)
4) **Platzbedarf** für die Speicherung von h (Programmtext, Parameter)
   (*gering* gegenüber n!)

Universelles Hashing (Idee): Bei der Festlegung der Hashfunktion h wird (kontrolliert) **Zufall** eingesetzt

5.3.2 Anspruchsvollere Hashfunktionen

Qualitätskriterien:

1) Zeit für **Auswertung** von $h(x)$ (*klein*!)
2) geringe **Wahrscheinlichkeit** für **Kollisionen**
   ohne Annahmen über die Schlüsselmenge S (ähnlich der zufälligen Situation!)
3) Effizientes **Generieren** einer neuen Hashfunktion
   (benötigt z. B. bei der Verdoppelungsstrategie)
4) **Platzbedarf** für die Speicherung von h (Programmtext, Parameter)
   (*gering* gegenüber n!)

Universelles Hashing (Idee): Bei der Festlegung der Hashfunktion h wird (kontrolliert) **Zufall** eingesetzt („Randomisierter Algorithmus“).

5.3.2 Anspruchsvollere Hashfunktionen

Qualitätskriterien:

1) Zeit für **Auswertung** von $h(x)$ (*klein*!)
2) geringe **Wahrscheinlichkeit** für **Kollisionen**
   ohne Annahmen über die Schlüsselmenge S (ähnlich der zufälligen Situation!)
3) Effizientes **Generieren** einer neuen Hashfunktion
   (benötigt z. B. bei der Verdoppelungsstrategie)
4) **Platzbedarf** für die Speicherung von h (Programmtext, Parameter)
   (*gering* gegenüber n!)

Universelles Hashing (Idee): Bei der Festlegung der Hashfunktion h wird (kontrolliert) **Zufall** eingesetzt („Randomisierter Algorithmus“).
(Bei Programmierung: „Pseudozufallszahlengenerator“, in jeder Prog.-Sprache bereitgestellt.)
Analyse beruht auf dadurch **vom Algorithmus erzwungenen** Zufallseigenschaften von h

5.3.2 Anspruchsvollere Hashfunktionen

Qualitätskriterien:

1) Zeit für **Auswertung** von $h(x)$ (*klein*!)
2) geringe **Wahrscheinlichkeit** für **Kollisionen**
   ohne Annahmen über die Schlüsselmenge S (ähnlich der zufälligen Situation!)
3) Effizientes **Generieren** einer neuen Hashfunktion
   (benötigt z. B. bei der Verdoppelungsstrategie)
4) **Platzbedarf** für die Speicherung von h (Programmtext, Parameter)
   (*gering* gegenüber n!)

Universelles Hashing (Idee): Bei der Festlegung der Hashfunktion h wird (kontrolliert) **Zufall** eingesetzt („Randomisierter Algorithmus“).
(Bei Programmierung: „Pseudozufallszahlengenerator“, in jeder Prog.-Sprache bereitgestellt.)
Analyse beruht auf dadurch **vom Algorithmus erzwungenen** Zufallseigenschaften von h, nicht auf Annahmen über S.

Beispiel: Alphabet $\Sigma = [s]$, $U \subseteq [s]^r$, und $m = \boldsymbol{p} \geq s$ eine **Primzahl**.

Beispiel: Alphabet $\Sigma = [s]$, $U \subseteq [s]^r$, und $m = p \geq s$ eine **Primzahl**.

[*Konkret*: $U = (\text{ISO 8859-1})^{20} = \{0, \ldots, 255\}^{20}$, $s = 256$, $m = p = 1009$.]

Beispiel: Alphabet $\Sigma = [s]$, $U \subseteq [s]^r$, und $m = \boldsymbol{p} \geq s$ eine **Primzahl**.

[*Konkret*: $U =$ (ISO 8859-1)$^{20} = \{0, \ldots, 255\}^{20}$, $s = 256$, $m = p = 1009$.]

$\boldsymbol{a_1, \ldots, a_r}$ seien (uniform) **zufällig gewählt** aus $[p]$. (In *empty*(p).)

Beispiel: Alphabet $\Sigma = [s]$, $U \subseteq [s]^r$, und $m = p \geq s$ eine **Primzahl**.

[*Konkret*: $U = (\text{ISO 8859-1})^{20} = \{0, \ldots, 255\}^{20}$, $s = 256$, $m = p = 1009$.]

$a_1, \ldots, a_r$ seien (uniform) **zufällig gewählt** aus $[p]$. (In *empty*(p).)

Definiere

$$h(\underbrace{c_1 \cdots c_r}_{x}) = (a_1 \cdot c_1 + \cdots + a_r \cdot c_r) \bmod p.$$

Beispiel: Alphabet $\Sigma = [s]$, $U \subseteq [s]^r$, und $m = p \geq s$ eine **Primzahl**.

[*Konkret*: $U = (\text{ISO 8859-1})^{20} = \{0, \ldots, 255\}^{20}$, $s = 256$, $m = p = 1009$.]

$a_1, \ldots, a_r$ seien (uniform) **zufällig gewählt** aus $[p]$. (In *empty*(p).)

Definiere

$$h(\underbrace{c_1 \cdots c_r}_{x}) = (a_1 \cdot c_1 + \cdots + a_r \cdot c_r) \bmod p.$$

Proposition 5.3.1

Beispiel: Alphabet $\Sigma = [s]$, $U \subseteq [s]^r$, und $m = p \geq s$ eine **Primzahl**.

[*Konkret*: $U = (\text{ISO 8859-1})^{20} = \{0, \ldots, 255\}^{20}$, $s = 256$, $m = p = 1009$.]

$a_1, \ldots, a_r$ seien (uniform) **zufällig gewählt** aus $[p]$. (In *empty*(p).)

Definiere

$$h(\underbrace{c_1 \cdots c_r}_{x}) = (a_1 \cdot c_1 + \cdots + a_r \cdot c_r) \bmod p.$$

Proposition 5.3.1

In der beschriebenen Situation gilt für $x, z \in U$ mit $x \neq z$:

Beispiel: Alphabet $\Sigma = [s]$, $U \subseteq [s]^r$, und $m = p \geq s$ eine **Primzahl**.

[*Konkret*: $U = (\text{ISO 8859-1})^{20} = \{0, \ldots, 255\}^{20}$, $s = 256$, $m = p = 1009$.]

$a_1, \ldots, a_r$ seien (uniform) **zufällig gewählt** aus $[p]$. (In *empty*(p).)

Definiere

$$h(\underbrace{c_1 \cdots c_r}_{x}) = (a_1 \cdot c_1 + \cdots + a_r \cdot c_r) \bmod p.$$

Proposition 5.3.1

In der beschriebenen Situation gilt für $x, z \in U$ mit $x \neq z$:

$$\mathbf{Pr}(h(x) = h(z)) = \frac{1}{p}.$$

Beispiel: Alphabet $\Sigma = [s]$, $U \subseteq [s]^r$, und $m = p \geq s$ eine **Primzahl**.

[*Konkret*: $U = (\text{ISO 8859-1})^{20} = \{0, \ldots, 255\}^{20}$, $s = 256$, $m = p = 1009$.]

$a_1, \ldots, a_r$ seien (uniform) **zufällig gewählt** aus $[p]$. (In *empty*(p).)

Definiere

$$h(\underbrace{c_1 \cdots c_r}_{x}) = (a_1 \cdot c_1 + \cdots + a_r \cdot c_r) \bmod p.$$

Proposition 5.3.1

In der beschriebenen Situation gilt für $x, z \in U$ mit $x \neq z$:

$$\mathbf{Pr}(h(x) = h(z)) = \frac{1}{p}.$$

(Zufall in der Wahrscheinlichkeit stammt aus dem Algorithmus.) **$(\mathbf{UF_2})$ ist erfüllt!**

Beweis: Sei $x = (c_1, \ldots, c_r)$, $z = (d_1, \ldots, d_r)$.

Beweis: Sei $x = (c_1, \ldots, c_r)$, $z = (d_1, \ldots, d_r)$.

Rechne im **Körper** $\mathbb{Z}_p = \mathbb{Z}/p\mathbb{Z}$ „=“ $[p]$ (also modulo p).

Beweis: Sei $x = (c_1, \ldots, c_r)$, $z = (d_1, \ldots, d_r)$.

Rechne im **Körper** $\mathbb{Z}_p = \mathbb{Z}/p\mathbb{Z}$ „=“ $[p]$ (also modulo p).

Es gibt p^r viele Möglichkeiten für $a = (a_1, \ldots, a_r)$.

Beweis: Sei $x = (c_1, \ldots, c_r)$, $z = (d_1, \ldots, d_r)$.

Rechne im **Körper** $\mathbb{Z}_p = \mathbb{Z}/p\mathbb{Z}$ „=“ $[p]$ (also modulo p).

Es gibt p^r viele Möglichkeiten für $a = (a_1, \ldots, a_r)$.

Wie viele davon erfüllen

$a_1c_1 + \ldots + a_rc_r = a_1d_1 + \ldots + a_rd_r$

Beweis: Sei $x = (c_1, \ldots, c_r)$, $z = (d_1, \ldots, d_r)$.

Rechne im **Körper** $\mathbb{Z}_p = \mathbb{Z}/p\mathbb{Z}$ „=“ $[p]$ (also modulo p).

Es gibt p^r viele Möglichkeiten für $a = (a_1, \ldots, a_r)$.

Wie viele davon erfüllen

$a_1c_1 + \ldots + a_rc_r = a_1d_1 + \ldots + a_rd_r$, d. h. $a_1(c_1 - d_1) + \ldots + a_r(c_r - d_r) = 0$?

Beweis: Sei $x = (c_1, \ldots, c_r)$, $z = (d_1, \ldots, d_r)$.

Rechne im **Körper** $\mathbb{Z}_p = \mathbb{Z}/p\mathbb{Z}$ „=“ $[p]$ (also modulo p).

Es gibt p^r viele Möglichkeiten für $a = (a_1, \ldots, a_r)$.

Wie viele davon erfüllen

$a_1c_1 + \ldots + a_rc_r = a_1d_1 + \ldots + a_rd_r$, d. h. $a_1(c_1 - d_1) + \ldots + a_r(c_r - d_r) = 0$?

Beispielsweise sei $c_r - d_r \neq 0$.

Beweis: Sei $x = (c_1, \ldots, c_r)$, $z = (d_1, \ldots, d_r)$.

Rechne im **Körper** $\mathbb{Z}_p = \mathbb{Z}/p\mathbb{Z}$ „=“ $[p]$ (also modulo p).

Es gibt p^r viele Möglichkeiten für $a = (a_1, \ldots, a_r)$.

Wie viele davon erfüllen

$a_1c_1 + \ldots + a_rc_r = a_1d_1 + \ldots + a_rd_r$, d. h. $a_1(c_1 - d_1) + \ldots + a_r(c_r - d_r) = 0$?

Beispielsweise sei $c_r - d_r \neq 0$.

Für jede Wahl von $a_1, \ldots, a_{r-1}$ gibt es genau ein passendes a_r, nämlich

Beweis: Sei $x = (c_1, \ldots, c_r)$, $z = (d_1, \ldots, d_r)$.

Rechne im **Körper** $\mathbb{Z}_p = \mathbb{Z}/p\mathbb{Z}$ „=“ $[p]$ (also modulo p).

Es gibt p^r viele Möglichkeiten für $a = (a_1, \ldots, a_r)$.

Wie viele davon erfüllen

$a_1c_1 + \ldots + a_rc_r = a_1d_1 + \ldots + a_rd_r$, d. h. $a_1(c_1 - d_1) + \ldots + a_r(c_r - d_r) = 0$?

Beispielsweise sei $c_r - d_r \neq 0$.

Für jede Wahl von $a_1, \ldots, a_{r-1}$ gibt es genau ein passendes a_r, nämlich

$$a_r = -(a_1(c_1 - d_1) + \ldots + a_{r-1}(c_{r-1} - d_{r-1})) \cdot (c_r - d_r)^{-1}.$$

Beweis: Sei $x = (c_1, \ldots, c_r)$, $z = (d_1, \ldots, d_r)$.

Rechne im **Körper** $\mathbb{Z}_p = \mathbb{Z}/p\mathbb{Z}$ „=“ $[p]$ (also modulo p).

Es gibt p^r viele Möglichkeiten für $a = (a_1, \ldots, a_r)$.

Wie viele davon erfüllen
$a_1c_1 + \ldots + a_rc_r = a_1d_1 + \ldots + a_rd_r$, d. h. $a_1(c_1 - d_1) + \ldots + a_r(c_r - d_r) = 0$?

Beispielsweise sei $c_r - d_r \neq 0$.

Für jede Wahl von $a_1, \ldots, a_{r-1}$ gibt es genau ein passendes a_r, nämlich

$$a_r = -(a_1(c_1 - d_1) + \ldots + a_{r-1}(c_{r-1} - d_{r-1})) \cdot (c_r - d_r)^{-1}.$$

(Wir rechnen modulo p, mit Inversen im Körper $\mathbb{Z}_p$.) Also ist die Wahrscheinlichkeit, einen passenden Vektor $(a_1, \ldots, a_r)$ zu wählen, genau $\frac{p^{r-1}}{p^r} = \frac{1}{p}$.

Beweis: Sei $x = (c_1, \ldots, c_r)$, $z = (d_1, \ldots, d_r)$.

Rechne im **Körper** $\mathbb{Z}_p = \mathbb{Z}/p\mathbb{Z}$ „=“ $[p]$ (also modulo p).

Es gibt p^r viele Möglichkeiten für $a = (a_1, \ldots, a_r)$.

Wie viele davon erfüllen
$a_1c_1 + \ldots + a_rc_r = a_1d_1 + \ldots + a_rd_r$, d. h. $a_1(c_1 - d_1) + \ldots + a_r(c_r - d_r) = 0$?

Beispielsweise sei $c_r - d_r \neq 0$.

Für jede Wahl von $a_1, \ldots, a_{r-1}$ gibt es genau ein passendes a_r, nämlich

$$a_r = -(a_1(c_1 - d_1) + \ldots + a_{r-1}(c_{r-1} - d_{r-1})) \cdot (c_r - d_r)^{-1}.$$

(Wir rechnen modulo p, mit Inversen im Körper $\mathbb{Z}_p$.) Also ist die Wahrscheinlichkeit, einen passenden Vektor $(a_1, \ldots, a_r)$ zu wählen, genau $\dfrac{p^{r-1}}{p^r} = \dfrac{1}{p}$. □

Variante: $U \subseteq [p]$, p Primzahl.

Variante: $U \subseteq [p]$, p Primzahl.

Tabellengröße $m \leq p$ beliebig.

Variante: $U \subseteq [p]$, p Primzahl.

Tabellengröße $m \leq p$ beliebig.

Wähle $\boldsymbol{a}$ aus $\{1, \ldots, p-1\}$ (uniform) **zufällig**.

Variante: $U \subseteq [p]$, p Primzahl.

Tabellengröße $m \leq p$ beliebig.

Wähle $\boldsymbol{a}$ aus $\{1, \ldots, p-1\}$ (uniform) **zufällig**.

Definiere

$$h(x) = ((\boldsymbol{a} \cdot x) \bmod p) \bmod m.$$

Variante: $U \subseteq [p]$, p Primzahl.

Tabellengröße $m \leq p$ beliebig.

Wähle $\boldsymbol{a}$ aus $\{1, \ldots, p-1\}$ (uniform) **zufällig**.

Definiere

$$h(x) = ((\boldsymbol{a} \cdot x) \bmod p) \bmod m.$$

Proposition 5.3.2

Variante: $U \subseteq [p]$, p Primzahl.

Tabellengröße $m \leq p$ beliebig.

Wähle $\boldsymbol{a}$ aus $\{1, \ldots, p-1\}$ (uniform) **zufällig**.

Definiere

$$h(x) = ((\boldsymbol{a} \cdot x) \bmod p) \bmod m.$$

Proposition 5.3.2

In dieser Situation gilt für $x, z \in U$ mit $x \neq z$:

Variante: $U \subseteq [p]$, p Primzahl.

Tabellengröße $m \leq p$ beliebig.

Wähle $\boldsymbol{a}$ aus $\{1, \ldots, p-1\}$ (uniform) **zufällig**.

Definiere

$$h(x) = ((\boldsymbol{a} \cdot x) \bmod p) \bmod m.$$

Proposition 5.3.2

In dieser Situation gilt für $x, z \in U$ mit $x \neq z$:

$$\mathbf{Pr}(h(x) = h(z)) \leq \frac{2}{m}.$$

Variante: $U \subseteq [p]$, p Primzahl.

Tabellengröße $m \leq p$ beliebig.

Wähle $\boldsymbol{a}$ aus $\{1, \ldots, p-1\}$ (uniform) **zufällig**.

Definiere

$$h(x) = ((\boldsymbol{a} \cdot x) \bmod p) \bmod m.$$

Proposition 5.3.2

In dieser Situation gilt für $x, z \in U$ mit $x \neq z$:

$$\mathbf{Pr}(h(x) = h(z)) \leq \frac{2}{m}.$$

(D. h. eine Abschwächung von (UF_2) ist erfüllt.)

Variante: $U \subseteq [p]$, p Primzahl.

Tabellengröße $m \leq p$ beliebig.

Wähle $\boldsymbol{a}$ aus $\{1, \ldots, p-1\}$ (uniform) **zufällig**.

Definiere

$$h(x) = ((\boldsymbol{a} \cdot x) \bmod p) \bmod m.$$

Proposition 5.3.2

In dieser Situation gilt für $x, z \in U$ mit $x \neq z$:

$$\mathbf{Pr}(h(x) = h(z)) \leq \frac{2}{m}.$$

(D. h. eine Abschwächung von (UF_2) ist erfüllt.)

Beweis (nicht prüfungsrelevant): Druckfolien.

Proposition 5.3.3

Proposition 5.3.3

Wenn man beim **multiplikativen Hashing** (wobei $U = [2^k]$, $m = 2^l$), mit

$$h(x) = (\boldsymbol{a} \cdot x \bmod 2^k) \operatorname{div} 2^{k-l}$$

Proposition 5.3.3

Wenn man beim **multiplikativen Hashing** (wobei $U = [2^k]$, $m = 2^l$), mit

$$h(x) = (\boldsymbol{a} \cdot x \bmod 2^k) \operatorname{div} 2^{k-l}$$

als $\boldsymbol{a}$ eine **zufällige** **ungerade** Zahl in $[2^k]$ wählt,

Proposition 5.3.3

Wenn man beim **multiplikativen Hashing** (wobei $U = [2^k]$, $m = 2^l$), mit

$$h(x) = (\boldsymbol{a} \cdot x \bmod 2^k) \operatorname{div} 2^{k-l}$$

als $\boldsymbol{a}$ eine **zufällige** **ungerade** Zahl in $[2^k]$ wählt, dann gilt für beliebige $x, z \in U$, $x \neq z$:

$$\mathbf{Pr}(h(x) = h(z)) \leq \frac{2}{2^l} = \frac{2}{m}.$$

Proposition 5.3.3

Wenn man beim **multiplikativen Hashing** (wobei $U = [2^k]$, $m = 2^l$), mit

$$h(x) = (\boldsymbol{a} \cdot x \bmod 2^k) \operatorname{div} 2^{k-l}$$

als $\boldsymbol{a}$ eine **zufällige** **ungerade** Zahl in $[2^k]$ wählt, dann gilt für beliebige $x, z \in U$, $x \neq z$:

$$\mathbf{Pr}(h(x) = h(z)) \leq \frac{2}{2^l} = \frac{2}{m}.$$

D. h.: Eine Abschwächung von (UF_2) ist erfüllt.

Proposition 5.3.3

Wenn man beim **multiplikativen Hashing** (wobei $U = [2^k]$, $m = 2^l$), mit

$$h(x) = (\boldsymbol{a} \cdot x \bmod 2^k) \operatorname{div} 2^{k-l}$$

als $\boldsymbol{a}$ eine **zufällige** **ungerade** Zahl in $[2^k]$ wählt, dann gilt für beliebige $x, z \in U$, $x \neq z$:

$$\mathbf{Pr}(h(x) = h(z)) \leq \frac{2}{2^l} = \frac{2}{m}.$$

D. h.: Eine Abschwächung von (UF_2) ist erfüllt.

(*Beweis*: Originalliteratur.)

Tabellenhashing [Zobrist, 1970]

Tabellenhashing [Zobrist, 1970]

Universum: $U = \Sigma^r$,

Tabellenhashing [Zobrist, 1970]

Universum: $U = \Sigma^r$, mit $\Sigma = \{0, \ldots, s-1\}$.

Tabellenhashing [Zobrist, 1970]

Universum: $U = \Sigma^r$, mit $\Sigma = \{0, \ldots, s-1\}$.

$m = 2^l$ mit $l \leq w$ (w: Wortlänge, z. B. $w = 32$).

Tabellenhashing [Zobrist, 1970]

Universum: $U = \Sigma^r$, mit $\Sigma = \{0, \ldots, s-1\}$.

$m = 2^l$ mit $l \leq w$ (w: Wortlänge, z. B. $w = 32$).

Wertebereich: $[m]$, entspricht $\{0,1\}^l$ (via Binärdarstellung!).

Tabellenhashing [Zobrist, 1970]

Universum: $U = \Sigma^r$, mit $\Sigma = \{0, \ldots, s-1\}$.

$m = 2^l$ mit $l \leq w$ (w: Wortlänge, z. B. $w = 32$).

Wertebereich: $[m]$, entspricht $\{0,1\}^l$ (via Binärdarstellung!).

Repräsentation der Hashfunktion h:

Tabellenhashing [Zobrist, 1970]

Universum: $U = \Sigma^r$, mit $\Sigma = \{0, \ldots, s-1\}$.

$m = 2^l$ mit $l \leq w$ (w: Wortlänge, z. B. $w = 32$).

Wertebereich: $[m]$, entspricht $\{0,1\}^l$ (via Binärdarstellung!).

Repräsentation der Hashfunktion h:

Array $\texttt{A}[1\,..\,r\,,0\,..\,s-1]$ mit Einträgen aus $\{0,1\}^w$.

Tabellenhashing [Zobrist, 1970]

Universum: $U = \Sigma^r$, mit $\Sigma = \{0, \ldots, s-1\}$.

$m = 2^l$ mit $l \leq w$ (w: Wortlänge, z. B. $w = 32$).

Wertebereich: $[m]$, entspricht $\{0,1\}^l$ (via Binärdarstellung!).

Repräsentation der Hashfunktion h:

Array $\texttt{A}[1\,..\,r\,,0\,..\,s-1]$ mit Einträgen aus $\{0,1\}^w$.

$h(c_1 \cdots c_r) =$ die ersten l Bits von $\texttt{A}[1,c_1] \oplus \cdots \oplus \texttt{A}[r,c_r]$

Tabellenhashing [Zobrist, 1970]

Universum: $U = \Sigma^r$, mit $\Sigma = \{0, \ldots, s-1\}$.

$m = 2^l$ mit $l \leq w$ (w: Wortlänge, z. B. $w = 32$).

Wertebereich: $[m]$, entspricht $\{0,1\}^l$ (via Binärdarstellung!).

Repräsentation der Hashfunktion h:

Array $\texttt{A}[1\,..\,r\,,0\,..\,s-1]$ mit Einträgen aus $\{0,1\}^w$.

$h(c_1 \cdots c_r) =$ die ersten l Bits von $\texttt{A}[1,c_1] \oplus \cdots \oplus \texttt{A}[r,c_r]$

Dabei: $\oplus$ ist bitweises XOR, auf ganzes Wort angewendet.

Tabellenhashing [Zobrist, 1970]

Universum: $U = \Sigma^r$, mit $\Sigma = \{0, \ldots, s-1\}$.

$m = 2^l$ mit $l \leq w$ (w: Wortlänge, z. B. $w = 32$).

Wertebereich: $[m]$, entspricht $\{0,1\}^l$ (via Binärdarstellung!).

Repräsentation der Hashfunktion h:

Array $\mathtt{A}[1\,..\,r\,,0\,..\,s-1]$ mit Einträgen aus $\{0,1\}^w$.

$h(c_1 \cdots c_r) =$ die ersten l Bits von $\mathtt{A}[1,c_1] \oplus \cdots \oplus \mathtt{A}[r,c_r]$

Dabei: $\oplus$ ist bitweises XOR, auf ganzes Wort angewendet.
(Maschinenoperation – effizient!)

Beispiel:

Beispiel: $\Sigma = \{0, 1, \ldots, 7\}$, $r = 6$ (1 Schlüssel $= 6$ Oktalziffern)

Beispiel: $\Sigma = \{0, 1, \ldots, 7\}$, $r = 6$ (1 Schlüssel $= 6$ Oktalziffern), $w = 4$, $l = 3$.

Beispiel: $\Sigma = \{0, 1, \ldots, 7\}$, $r = 6$ (1 Schlüssel $= 6$ Oktalziffern), $w = 4$, $l = 3$.

Array `A`:

	Buchstaben in Σ (Oktalziffern)							
	0	1	2	3	4	5	6	7
Position $i = 1$	0010	1001	0011	1010	0111	0000	1101	0111
2	0011	1000	1111	0100	1001	1101	0110	0110
3	1000	0110	0101	1001	1110	1111	0000	1110
4	1101	0100	1110	0111	1101	0100	1010	0110
5	1100	1001	1100	1011	0101	1011	1110	0000
6	0110	1101	1001	1111	1000	1110	0011	0111

Beispiel: $\Sigma = \{0, 1, \ldots, 7\}$, $r = 6$ (1 Schlüssel = 6 Oktalziffern), $w = 4$, $l = 3$.

Array `A`:

	Buchstaben in Σ (Oktalziffern)							
	0	1	2	3	4	5	6	7
Position $i = 1$	0010	1001	0011	1010	0111	0000	1101	0111
2	0011	1000	1111	0100	1001	1101	0110	0110
3	1000	0110	0101	1001	1110	1111	0000	1110
4	1101	0100	1110	0111	1101	0100	1010	0110
5	1100	1001	1100	1011	0101	1011	1110	0000
6	0110	1101	1001	1111	1000	1110	0011	0111

$h(3\,5\,3\,7\,4\,3)$

Beispiel: $\Sigma = \{0, 1, \ldots, 7\}$, $r = 6$ (1 Schlüssel = 6 Oktalziffern), $w = 4$, $l = 3$.

Array `A`:

	Buchstaben in Σ (Oktalziffern)							
	0	1	2	3	4	5	6	7
Position $i = 1$	0010	1001	0011	1010	0111	0000	1101	0111
2	0011	1000	1111	0100	1001	1101	0110	0110
3	1000	0110	0101	1001	1110	1111	0000	1110
4	1101	0100	1110	0111	1101	0100	1010	0110
5	1100	1001	1100	1011	0101	1011	1110	0000
6	0110	1101	1001	1111	1000	1110	0011	0111

$h(3\,5\,3\,7\,4\,3)$
$=$ die ersten 3 Bits von

Beispiel: $\Sigma = \{0, 1, \ldots, 7\}$, $r = 6$ (1 Schlüssel = 6 Oktalziffern), $w = 4$, $l = 3$.

Array `A`:

	Buchstaben in Σ (Oktalziffern)							
	0	1	2	3	4	5	6	7
Position $i = 1$	0010	1001	0011	1010	0111	0000	1101	0111
2	0011	1000	1111	0100	1001	1101	0110	0110
3	1000	0110	0101	1001	1110	1111	0000	1110
4	1101	0100	1110	0111	1101	0100	1010	0110
5	1100	1001	1100	1011	0101	1011	1110	0000
6	0110	1101	1001	1111	1000	1110	0011	0111

$h(3\,5\,3\,7\,4\,3)$
$=$ die ersten 3 Bits von 1010

Beispiel: $\Sigma = \{0, 1, \ldots, 7\}$, $r = 6$ (1 Schlüssel = 6 Oktalziffern), $w = 4$, $l = 3$.

Array `A`:

	Buchstaben in Σ (Oktalziffern)							
	0	1	2	3	4	5	6	7
Position $i = 1$	0010	1001	0011	1010	0111	0000	1101	0111
2	0011	1000	1111	0100	1001	1101	0110	0110
3	1000	0110	0101	1001	1110	1111	0000	1110
4	1101	0100	1110	0111	1101	0100	1010	0110
5	1100	1001	1100	1011	0101	1011	1110	0000
6	0110	1101	1001	1111	1000	1110	0011	0111

$h(3\,5\,3\,7\,4\,3)$
$=$ die ersten 3 Bits von $1010 \oplus 1101$

Beispiel: $\Sigma = \{0, 1, \ldots, 7\}$, $r = 6$ (1 Schlüssel = 6 Oktalziffern), $w = 4$, $l = 3$.

Array `A`:

	Buchstaben in Σ (Oktalziffern)							
	0	1	2	3	4	5	6	7
Position $i = 1$	0010	1001	0011	1010	0111	0000	1101	0111
2	0011	1000	1111	0100	1001	1101	0110	0110
3	1000	0110	0101	1001	1110	1111	0000	1110
4	1101	0100	1110	0111	1101	0100	1010	0110
5	1100	1001	1100	1011	0101	1011	1110	0000
6	0110	1101	1001	1111	1000	1110	0011	0111

$h(3\,5\,3\,7\,4\,3)$
$=$ die ersten 3 Bits von $1010 \oplus 1101 \oplus 1001$

Beispiel: $\Sigma = \{0, 1, \ldots, 7\}$, $r = 6$ (1 Schlüssel = 6 Oktalziffern), $w = 4$, $l = 3$.

Array `A`:

	Buchstaben in Σ (Oktalziffern)							
	0	1	2	3	4	5	6	7
Position $i = 1$	0010	1001	0011	1010	0111	0000	1101	0111
2	0011	1000	1111	0100	1001	1101	0110	0110
3	1000	0110	0101	1001	1110	1111	0000	1110
4	1101	0100	1110	0111	1101	0100	1010	0110
5	1100	1001	1100	1011	0101	1011	1110	0000
6	0110	1101	1001	1111	1000	1110	0011	0111

$h(3\,5\,3\,7\,4\,3)$
$=$ die ersten 3 Bits von $1010 \oplus 1101 \oplus 1001 \oplus 0110$

Beispiel: $\Sigma = \{0, 1, \ldots, 7\}$, $r = 6$ (1 Schlüssel = 6 Oktalziffern), $w = 4$, $l = 3$.

Array `A`:

	Buchstaben in Σ (Oktalziffern)							
	0	1	2	3	4	5	6	7
Position $i = 1$	0010	1001	0011	1010	0111	0000	1101	0111
2	0011	1000	1111	0100	1001	1101	0110	0110
3	1000	0110	0101	1001	1110	1111	0000	1110
4	1101	0100	1110	0111	1101	0100	1010	0110
5	1100	1001	1100	1011	0101	1011	1110	0000
6	0110	1101	1001	1111	1000	1110	0011	0111

$h(3\,5\,3\,7\,4\,3)$
$=$ die ersten 3 Bits von $1010 \oplus 1101 \oplus 1001 \oplus 0110 \oplus 0101$

Beispiel: $\Sigma = \{0, 1, \ldots, 7\}$, $r = 6$ (1 Schlüssel = 6 Oktalziffern), $w = 4$, $l = 3$.

Array `A`:

	Buchstaben in Σ (Oktalziffern)							
	0	1	2	3	4	5	6	7
Position $i = 1$	0010	1001	0011	1010	0111	0000	1101	0111
2	0011	1000	1111	0100	1001	1101	0110	0110
3	1000	0110	0101	1001	1110	1111	0000	1110
4	1101	0100	1110	0111	1101	0100	1010	0110
5	1100	1001	1100	1011	0101	1011	1110	0000
6	0110	1101	1001	1111	1000	1110	0011	0111

$h(3\,5\,3\,7\,4\,3)$

$=$ die ersten 3 Bits von $1010 \oplus 1101 \oplus 1001 \oplus 0110 \oplus 0101 \oplus 1111$

Beispiel: $\Sigma = \{0, 1, \ldots, 7\}$, $r = 6$ (1 Schlüssel = 6 Oktalziffern), $w = 4$, $l = 3$.

Array `A`:

	Buchstaben in Σ (Oktalziffern)							
	0	1	2	3	4	5	6	7
Position $i = 1$	0010	1001	0011	1010	0111	0000	1101	0111
2	0011	1000	1111	0100	1001	1101	0110	0110
3	1000	0110	0101	1001	1110	1111	0000	1110
4	1101	0100	1110	0111	1101	0100	1010	0110
5	1100	1001	1100	1011	0101	1011	1110	0000
6	0110	1101	1001	1111	1000	1110	0011	0111

$h(3\,5\,3\,7\,4\,3)$
$=$ die ersten 3 Bits von $1010 \oplus 1101 \oplus 1001 \oplus 0110 \oplus 0101 \oplus 1111$
$=$ die ersten 3 Bits von 0010

Beispiel: $\Sigma = \{0, 1, \ldots, 7\}$, $r = 6$ (1 Schlüssel = 6 Oktalziffern), $w = 4$, $l = 3$.

Array `A`:

	Buchstaben in Σ (Oktalziffern)							
	0	1	2	3	4	5	6	7
Position $i = 1$	0010	1001	0011	1010	0111	0000	1101	0111
2	0011	1000	1111	0100	1001	1101	0110	0110
3	1000	0110	0101	1001	1110	1111	0000	1110
4	1101	0100	1110	0111	1101	0100	1010	0110
5	1100	1001	1100	1011	0101	1011	1110	0000
6	0110	1101	1001	1111	1000	1110	0011	0111

$h(3\,5\,3\,7\,4\,3)$
$=$ die ersten 3 Bits von $1010 \oplus 1101 \oplus 1001 \oplus 0110 \oplus 0101 \oplus 1111$
$=$ die ersten 3 Bits von $0010 = 001$.

Nachteil: Array $\texttt{A}[1\,..\,r\,,0\,..\,s-1]$ benötigt **Platz** $r \cdot s$ (Wörter).

Nachteil: Array $\texttt{A}[1\,..\,r\,,0\,..\,s-1]$ benötigt **Platz** $r \cdot s$ (Wörter).

Beispiel: $r = 20$ ISO-8859-1-Zeichen pro Wort, $\Sigma = \{0,1\}^8$, $w = 32$.

Nachteil: Array $\texttt{A}[1\,..\,r\,,0\,..\,s-1]$ benötigt **Platz** $r \cdot s$ (Wörter).

Beispiel: $r = 20$ ISO-8859-1-Zeichen pro Wort, $\Sigma = \{0,1\}^8$, $w = 32$.
$\texttt{A}[1\,..\,r\,,0\,..\,s-1]$ benötigt
$rs(w/8) = 20 \cdot 256 \cdot 4 = 20480$ Bytes,

Nachteil: Array $\texttt{A}[1\,..\,r\,,0\,..\,s-1]$ benötigt **Platz** $r \cdot s$ (Wörter).

Beispiel: $r = 20$ ISO-8859-1-Zeichen pro Wort, $\Sigma = \{0,1\}^8$, $w = 32$.
$\texttt{A}[1\,..\,r\,,0\,..\,s-1]$ benötigt
$rs(w/8) = 20 \cdot 256 \cdot 4 = 20480$ Bytes, d. h. 20 KB.

Nachteil: Array $\texttt{A}[1\,..\,r\,,0\,..\,s-1]$ benötigt **Platz** $r \cdot s$ (Wörter).

Beispiel: $r = 20$ ISO-8859-1-Zeichen pro Wort, $\Sigma = \{0,1\}^8$, $w = 32$.
$\texttt{A}[1\,..\,r\,,0\,..\,s-1]$ benötigt
$rs(w/8) = 20 \cdot 256 \cdot 4 = 20480$ Bytes, d. h. 20 KB.
Sinnvoll nur für **große** Hashtabellen.

Nachteil: Array $\texttt{A}[1\,..\,r\,,0\,..\,s-1]$ benötigt **Platz** $r \cdot s$ (Wörter).

Beispiel: $r = 20$ ISO-8859-1-Zeichen pro Wort, $\Sigma = \{0,1\}^8$, $w = 32$.
$\texttt{A}[1\,..\,r\,,0\,..\,s-1]$ benötigt
$rs(w/8) = 20 \cdot 256 \cdot 4 = 20480$ Bytes, d. h. 20 KB.
Sinnvoll nur für **große** Hashtabellen.

Platzsparend, etwas langsamer:
$\Sigma = \{0,1\}^4 \mathrel{\widehat{=}} \{\texttt{0}, \texttt{1}, \ldots, \texttt{9}, \texttt{A}, \ldots, \texttt{F}\}$. („Tetraden“ bzw. Hexadezimalziffern.)

Platz im Beispiel: $rs(w/8) = 40 \cdot 16 \cdot 4 = 2560$ Bytes.

Vorteile von Tabellenhashing:

Vorteile von Tabellenhashing:

- **Extrem schnelle Auswertung** ($\oplus$ ebenso schnell wie $+$).

Vorteile von Tabellenhashing:

- **Extrem schnelle Auswertung** ($\oplus$ ebenso schnell wie $+$).
- Dasselbe `A` kann für $m = 2^4, 2^5, 2^6, \ldots, 2^w$ benutzt werden.

Vorteile von Tabellenhashing:

- **Extrem schnelle Auswertung** ($\oplus$ ebenso schnell wie $+$).
- Dasselbe `A` kann für $m = 2^4, 2^5, 2^6, \ldots, 2^w$ benutzt werden.

  Ideal für die Kombination mit der Verdoppelungsstrategie:

Vorteile von Tabellenhashing:

- **Extrem schnelle Auswertung** ($\oplus$ ebenso schnell wie $+$).
- Dasselbe `A` kann für $m = 2^4, 2^5, 2^6, \ldots, 2^w$ benutzt werden.

  Ideal für die Kombination mit der Verdoppelungsstrategie: Verdopple Tabellengröße $\widehat{=}$ erhöhe l um 1; Einträge in `A` bleiben unverändert

Vorteile von Tabellenhashing:

- **Extrem schnelle Auswertung** ($\oplus$ ebenso schnell wie $+$).
- Dasselbe `A` kann für $m = 2^4, 2^5, 2^6, \ldots, 2^w$ benutzt werden.

  Ideal für die Kombination mit der Verdoppelungsstrategie: Verdopple Tabellengröße $\widehat{=}$ erhöhe l um 1; Einträge in `A` bleiben unverändert.
- Beweisbar **gutes Verhalten**, wenn Einträge von `A` zufällig.

Vorteile von Tabellenhashing:

- **Extrem schnelle Auswertung** ($\oplus$ ebenso schnell wie $+$).
- Dasselbe `A` kann für $m = 2^4, 2^5, 2^6, \ldots, 2^w$ benutzt werden.

  Ideal für die Kombination mit der Verdoppelungsstrategie: Verdopple Tabellengröße $\widehat{=}$ erhöhe l um 1; Einträge in `A` bleiben unverändert.
- Beweisbar **gutes Verhalten**, wenn Einträge von `A` zufällig.

Proposition 5.3.4

Vorteile von Tabellenhashing:

- **Extrem schnelle Auswertung** ($\oplus$ ebenso schnell wie $+$).
- Dasselbe `A` kann für $m = 2^4, 2^5, 2^6, \ldots, 2^w$ benutzt werden.

  Ideal für die Kombination mit der Verdoppelungsstrategie: Verdopple Tabellengröße $\widehat{=}$ erhöhe l um 1; Einträge in `A` bleiben unverändert.
- Beweisbar **gutes Verhalten**, wenn Einträge von `A` zufällig.

Proposition 5.3.4

Wenn die Einträge in Array `A`$[1\,..\,r\,,0\,..\,s-1]$ uniform **zufällig** gewählt werden, dann gilt für $x, y, z \in U$ mit x, y, z verschieden und $a, b, c \in \{0,1\}^l$ beliebig:

Vorteile von Tabellenhashing:

- **Extrem schnelle Auswertung** ($\oplus$ ebenso schnell wie $+$).
- Dasselbe `A` kann für $m = 2^4, 2^5, 2^6, \ldots, 2^w$ benutzt werden.

  Ideal für die Kombination mit der Verdoppelungsstrategie: Verdopple Tabellengröße $\widehat{=}$ erhöhe l um 1; Einträge in `A` bleiben unverändert.
- Beweisbar **gutes Verhalten**, wenn Einträge von `A` zufällig.

Proposition 5.3.4

Wenn die Einträge in Array `A`$[1\,..\,r\,,0\,..\,s-1]$ uniform **zufällig** gewählt werden, dann gilt für $x, y, z \in U$ mit x, y, z verschieden und $a, b, c \in \{0,1\}^l$ beliebig:

$$\mathbf{Pr}(h(x) = a \wedge h(y) = b \wedge h(z) = c) = \tfrac{1}{m^3}.$$

Vorteile von Tabellenhashing:

- **Extrem schnelle Auswertung** ($\oplus$ ebenso schnell wie $+$).
- Dasselbe `A` kann für $m = 2^4, 2^5, 2^6, \ldots, 2^w$ benutzt werden.

  Ideal für die Kombination mit der Verdoppelungsstrategie: Verdopple Tabellengröße $\widehat{=}$ erhöhe l um 1; Einträge in `A` bleiben unverändert.
- Beweisbar **gutes Verhalten**, wenn Einträge von `A` zufällig.

Proposition 5.3.4

Wenn die Einträge in Array `A`$[1\,..\,r\,,0\,..\,s-1]$ uniform **zufällig** gewählt werden, dann gilt für $x, y, z \in U$ mit x, y, z verschieden und $a, b, c \in \{0,1\}^l$ beliebig:

$$\mathbf{Pr}(h(x) = a \wedge h(y) = b \wedge h(z) = c) = \tfrac{1}{m^3}.$$

Insbesondere gilt (UF_2).

Vorteile von Tabellenhashing:

- **Extrem schnelle Auswertung** ($\oplus$ ebenso schnell wie $+$).
- Dasselbe `A` kann für $m = 2^4, 2^5, 2^6, \ldots, 2^w$ benutzt werden.

  Ideal für die Kombination mit der Verdoppelungsstrategie: Verdopple Tabellengröße $\widehat{=}$ erhöhe l um 1; Einträge in `A` bleiben unverändert.
- Beweisbar **gutes Verhalten**, wenn Einträge von `A` zufällig.

Proposition 5.3.4

Wenn die Einträge in Array `A`$[1\,..\,r\,,0\,..\,s-1]$ uniform **zufällig** gewählt werden, dann gilt für $x, y, z \in U$ mit x, y, z verschieden und $a, b, c \in \{0,1\}^l$ beliebig:

$$\mathbf{Pr}(h(x) = a \wedge h(y) = b \wedge h(z) = c) = \tfrac{1}{m^3}.$$

Insbesondere gilt (UF_2).

Beweis: Übung.

Leichte Modifikation des Beweises von Satz 5.2.3. liefert:

Leichte Modifikation des Beweises von Satz 5.2.3. liefert:

Proposition 5.3.5

Sei c konstant. Wenn $\mathbf{Pr}(h(x) = h(z)) \leq \frac{c}{m}$,

Leichte Modifikation des Beweises von Satz 5.2.3. liefert:

Proposition 5.3.5

Sei c konstant. Wenn $\mathbf{Pr}(h(x) = h(z)) \leq \frac{c}{m}$, für beliebige $x, z \in U$, $x \neq z$,

Leichte Modifikation des Beweises von Satz 5.2.3. liefert:

Proposition 5.3.5

Sei c konstant. Wenn $\mathbf{Pr}(h(x) = h(z)) \leq \frac{c}{m}$, für beliebige $x, z \in U$, $x \neq z$, und wenn wir **Hashing mit verketteten Listen** benutzen, dann ist die **erwartete** Zahl von Schlüsselvergleichen

Leichte Modifikation des Beweises von Satz 5.2.3. liefert:

Proposition 5.3.5

Sei c konstant. Wenn $\mathbf{Pr}(h(x) = h(z)) \leq \frac{c}{m}$, für beliebige $x, z \in U$, $x \neq z$, und wenn wir **Hashing mit verketteten Listen** benutzen, dann ist die **erwartete** Zahl von Schlüsselvergleichen

- höchstens $c \cdot \alpha$ bei der **erfolglosen Suche**

Leichte Modifikation des Beweises von Satz 5.2.3. liefert:

Proposition 5.3.5

Sei c konstant. Wenn $\mathbf{Pr}(h(x) = h(z)) \leq \frac{c}{m}$, für beliebige $x, z \in U$, $x \neq z$, und wenn wir **Hashing mit verketteten Listen** benutzen, dann ist die **erwartete** Zahl von Schlüsselvergleichen

- höchstens $c \cdot \alpha$ bei der **erfolglosen Suche** und
- höchstens $1 + c \cdot \alpha$ bei der **erfolgreichen Suche**.

Leichte Modifikation des Beweises von Satz 5.2.3. liefert:

Proposition 5.3.5

Sei c konstant. Wenn $\mathbf{Pr}(h(x) = h(z)) \leq \frac{c}{m}$, für beliebige $x, z \in U$, $x \neq z$, und wenn wir **Hashing mit verketteten Listen** benutzen, dann ist die **erwartete** Zahl von Schlüsselvergleichen

- höchstens $c \cdot \alpha$ bei der **erfolglosen Suche** und
- höchstens $1 + c \cdot \alpha$ bei der **erfolgreichen Suche**.

Wenn $\alpha \leq \alpha_1$ sichergestellt ist, für α_1 konstant (Verdoppelungsstrategie):
Erwartete Suchzeit ist $\boldsymbol{O(1)}$.

Leichte Modifikation des Beweises von Satz 5.2.3. liefert:

Proposition 5.3.5

Sei c konstant. Wenn $\mathbf{Pr}(h(x) = h(z)) \leq \frac{c}{m}$, für beliebige $x, z \in U$, $x \neq z$, und wenn wir **Hashing mit verketteten Listen** benutzen, dann ist die **erwartete** Zahl von Schlüsselvergleichen

- höchstens $c \cdot \alpha$ bei der **erfolglosen Suche** und
- höchstens $1 + c \cdot \alpha$ bei der **erfolgreichen Suche**.

Wenn $\alpha \leq \alpha_1$ sichergestellt ist, für α_1 konstant (Verdoppelungsstrategie):
Erwartete Suchzeit ist $\boldsymbol{O(1)}$.

Stichwort: „Universelles Hashing“: siehe Druckfolien.

Vorlesungsvideo:

Geschlossenes Hashing mit Sondierungsstrategien

5.4 Geschlossenes Hashing

j	x
0	`September`$^{(0)}$
1	
2	`Maerz`$^{(2)}$ `April`$^{(2)}$
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`$^{(12)}$

$h(c_1 \dots c_r) = (\mathsf{num}(c_3) + 11) \bmod 13$, „Wunschplätze“

5.4 Geschlossenes Hashing

j	x
0	September$^{(0)}$
1	
2	Maerz$^{(2)}$ April$^{(2)}$
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$^{(12)}$

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

Kollisionen!

Sequentielles Einfügen, „lineares Sondieren“:

Sequentielles Einfügen, „lineares Sondieren“:

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

Sequentielles Einfügen, „lineares Sondieren“:

Platz	Eintrag
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	`Januar`$^{(11)}$
12	

Sequentielles Einfügen, „lineares Sondieren“:

Platz	Eintrag
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	`Januar`$^{(11)}$
12	`Februar`$^{(12)}$

Sequentielles Einfügen, „lineares Sondieren“:

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

Sequentielles Einfügen, „lineares Sondieren“:

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

Sequentielles Einfügen, „lineares Sondieren“:

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

Sequentielles Einfügen, „lineares Sondieren“:

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

Sequentielles Einfügen, „lineares Sondieren“:

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

Sequentielles Einfügen, „lineares Sondieren“:

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

Sequentielles Einfügen, „lineares Sondieren“:

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

Sequentielles Einfügen, „lineares Sondieren“:

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

Sequentielles Einfügen, „lineares Sondieren“:

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

Sequentielles Einfügen, „lineares Sondieren“:

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

Sequentielles Einfügen, „lineares Sondieren“:

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

Sequentielles Einfügen, „lineares Sondieren“:

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

Sequentielles Einfügen, „lineares Sondieren“:

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

Sequentielles Einfügen, „lineares Sondieren“:

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

Sequentielles Einfügen, „lineares Sondieren“:

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

Sequentielles Einfügen, „lineares Sondieren“:

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

5.4 Geschlossenes Hashing

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

5.4 Geschlossenes Hashing

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

Idee: Für jeden Schlüssel $x \in U$ gibt es eine
Sondierungsfolge $h(x,0), h(x,1), h(x,2), \ldots$ in $[m]$.

5.4 Geschlossenes Hashing

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

Idee: Für jeden Schlüssel $x \in U$ gibt es eine
Sondierungsfolge $h(x,0), h(x,1), h(x,2), \ldots$ 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.

5.4 Geschlossenes Hashing

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

Idee: Für jeden Schlüssel $x \in U$ gibt es eine
Sondierungsfolge $h(x,0), h(x,1), h(x,2), \ldots$ 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:

5.4 Geschlossenes Hashing

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

Idee: Für jeden Schlüssel $x \in U$ gibt es eine
Sondierungsfolge $h(x,0), h(x,1), h(x,2), \ldots$ 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:

(*)

5.4 Geschlossenes Hashing

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

Idee: Für jeden Schlüssel $x \in U$ gibt es eine
Sondierungsfolge $h(x,0), h(x,1), h(x,2), \ldots$ 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:

(∗) $h(x,0), h(x,1), \ldots, h(x,m-1)$ erreicht jede Zelle,

5.4 Geschlossenes Hashing

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

Idee: Für jeden Schlüssel $x \in U$ gibt es eine
Sondierungsfolge $h(x,0), h(x,1), h(x,2), \ldots$ 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:

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

5.4 Geschlossenes Hashing

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

Idee: Für jeden Schlüssel $x \in U$ gibt es eine
Sondierungsfolge $h(x,0), h(x,1), h(x,2), \ldots$ 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:

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

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

A – Lineares Sondieren

A – Lineares Sondieren $h(x,k) = (h(x) + k) \bmod m,\ k = 0, 1, 2, \ldots$

A – Lineares Sondieren $h(x,k) = (h(x) + k) \bmod m,\ k = 0, 1, 2, \ldots$

Offensichtlich: $(*)$ erfüllt.

A – Lineares Sondieren $h(x,k) = (h(x) + k) \bmod m,\ k = 0, 1, 2, \ldots$

Offensichtlich: $(*)$ erfüllt.

Beispiel: Schon gesehen.

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

empty(m): Lege leeres Array $\texttt{T}[0\,..\,m-1]$ an.

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

empty(m): Lege leeres Array $\mathtt{T}[0\,..\,m-1]$ an.

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

Wähle Hashfunktion h.
$\mathtt{T}[j] \leftarrow$ *leer*, für $j \in [m]$.

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

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

Wähle Hashfunktion h.

`T`$[j] \leftarrow$ *leer*, für $j \in [m]$.

`load` $\leftarrow 0$, initialisiere `maxload` mit Wert $< m$.

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

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

Wähle Hashfunktion h.
`T`$[j] \leftarrow$ *leer*, für $j \in [m]$.
`load` $\leftarrow 0$, initialisiere `maxload` mit Wert $< m$.

insert(x, r): $j \leftarrow h(x)$;
Finde erstes l in der Folge j, $(j+1) \bmod m$, $(j+2) \bmod m$, . . .
mit `T`$[l]$`.key` $= x$ **oder** `T`$[l] =$ *leer*.

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

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

Wähle Hashfunktion h.
`T`$[j] \leftarrow$ *leer*, für $j \in [m]$.
`load` $\leftarrow 0$, initialisiere `maxload` mit Wert $< m$.

insert(x, r): $j \leftarrow h(x)$;
Finde erstes l in der Folge j, $(j+1) \bmod m$, $(j+2) \bmod m$, . . .
mit `T`$[l]$`.key` $= x$ **oder** `T`$[l] =$ *leer*.
Im Fall `T`$[l]$`.key` $= x$: // **erfolgreiche Suche**, Update-Situation
`T`$[l]$`.data` $\leftarrow r$;

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

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

Wähle Hashfunktion h.
`T`$[j] \leftarrow$ *leer*, für $j \in [m]$.
`load` $\leftarrow 0$, initialisiere `maxload` mit Wert $< m$.

insert(x, r): $j \leftarrow h(x)$;
Finde erstes l in der Folge j, $(j+1) \bmod m$, $(j+2) \bmod m$, . . .
mit `T`$[l]$`.key` $= x$ **oder** `T`$[l] =$ *leer*.
Im Fall `T`$[l]$`.key` $= x$: // **erfolgreiche Suche**, Update-Situation
`T`$[l]$`.data` $\leftarrow r$;
Im Fall `T`$[l] =$ *leer*: // **erfolglose Suche**, neuer Eintrag

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

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

Wähle Hashfunktion h.

`T`$[j] \leftarrow$ *leer*, für $j \in [m]$.

`load` $\leftarrow 0$, initialisiere `maxload` mit Wert $< m$.

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

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

mit `T`$[l]$`.key` $= x$ **oder** `T`$[l] =$ *leer*.

Im Fall `T`$[l]$`.key` $= x$: // **erfolgreiche Suche**, Update-Situation

`T`$[l]$`.data` $\leftarrow r$;

Im Fall `T`$[l] =$ *leer*: // **erfolglose Suche**, neuer Eintrag

Falls `load` $=$ `maxload`: „Überlaufbehandlung“

// z. B. Übertragen der Einträge von `T` in neue, doppelt so große Tabelle

// Neustart von **insert**(x, r)

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

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

Wähle Hashfunktion h.

`T`$[j] \leftarrow$ *leer*, für $j \in [m]$.

`load` $\leftarrow 0$, initialisiere `maxload` mit Wert $< m$.

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

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

mit `T`$[l]$`.key` $= x$ **oder** `T`$[l] =$ *leer*.

Im Fall `T`$[l]$`.key` $= x$: // **erfolgreiche Suche**, Update-Situation

`T`$[l]$`.data` $\leftarrow r$;

Im Fall `T`$[l] =$ *leer*: // **erfolglose Suche**, neuer Eintrag

Falls `load` $=$ `maxload`: „Überlaufbehandlung“

// z. B. Übertragen der Einträge von `T` in neue, doppelt so große Tabelle

// Neustart von **insert**(x, r)

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

lookup(x): $j \leftarrow h(x)$;
Finde erstes l in der Folge $h(x, j)$, $j = 0, 1, 2, \ldots$, mit `T[`l`].key` $= x$ **oder** `T[`l`]` $=$ *leer*.

Korrektheit der Suche: Aus der Invarianten folgt sofort durch Betrachten der Algorithmen, dass **lookup**(x) einen Eintrag mit Schlüssel x genau dann findet, wenn es einen gibt, und dass dieser immer den Datensatz r enthält, der von der neuesten Operation **insert**(x, r) herrührt. (Details: Druckfolien.)

lookup(x): $j \leftarrow h(x)$;
Finde erstes l in der Folge $h(x, j)$, $j = 0, 1, 2, \ldots$, mit `T[`l`].key` $= x$ **oder** `T[`l`]` $=$ *leer*.
Im Fall `T[`l`].key` $= x$: **return** `T[`l`].data`; // **erfolgreiche Suche**

Korrektheit der Suche: Aus der Invarianten folgt sofort durch Betrachten der Algorithmen, dass **lookup**(x) einen Eintrag mit Schlüssel x genau dann findet, wenn es einen gibt, und dass dieser immer den Datensatz r enthält, der von der neuesten Operation **insert**(x, r) herrührt. (Details: Druckfolien.)

lookup(x): $j \leftarrow h(x)$;
Finde erstes l in der Folge $h(x, j)$, $j = 0, 1, 2, \ldots$, 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: Aus der Invarianten folgt sofort durch Betrachten der Algorithmen, dass **lookup**(x) einen Eintrag mit Schlüssel x genau dann findet, wenn es einen gibt, und dass dieser immer den Datensatz r enthält, der von der neuesten Operation **insert**(x, r) herrührt. (Details: Druckfolien.)

lookup(x): $j \leftarrow h(x)$;
Finde erstes l in der Folge $h(x, j)$, $j = 0, 1, 2, \ldots$, 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**

Um die **Korrektheit** (des Ein-/Ausgabeverhaltens) zu zeigen, betrachten wir eine beliebige Operationenfolge und zeigen die folgende Invariante, für jeden Schlüssel x:

Invariante: Solange x nicht eingefügt wurde, gibt es in der Tabelle keinen Eintrag mit Schlüssel x. Nachdem x eingefügt wurde, gilt: Es gibt genau einen solchen Eintrag, in einer Zelle `T[`i`]`. Diese enthält immer den Datensatz r von der *letzten* Operation **insert**(x, r). Wenn $i = h(x, l)$, enthalten alle Zellen `T[`$h(x, j)$`]`, $0 \leq j < l$, einen Schlüssel $\neq x$. (Beweis durch Induktion über die ausgeführten Einfügungen, auf den Druckfolien.)

Korrektheit der Suche: Aus der Invarianten folgt sofort durch Betrachten der Algorithmen, dass **lookup**(x) einen Eintrag mit Schlüssel x genau dann findet, wenn es einen gibt, und dass dieser immer den Datensatz r enthält, der von der neuesten Operation **insert**(x, r) herrührt. (Details: Druckfolien.)

Beweis der Invarianten: Dass es keinen Eintrag mit Schlüssel x gibt, wenn x noch nicht eingefügt wurde, ist klar. Da Schlüssel nie gelöscht werden, sondern nur Datensätze überschrieben werden, bleibt ein einmal generierter Eintrag mit Schlüssel x immer erhalten.
Für den zweiten Teil benutzen wir Induktion. Betrachte die erste Einfügeoperation **insert**(x, r) für x. Die Suche nach x endet in einer leeren Zelle `T[`i`]`, mit $i = h(x, l)$, wobei alle Zellen `T[`$h(x, j)$`]`, $0 \leq j < l$, einen Schlüssel $\neq x$ enthalten. Hier wird (x, r) gespeichert. Wenn später in **insert**(x, r') erneut nach x gesucht wird, dann folgt aus der Invarianten und dem Suchverfahren, dass die eindeutig bestimmte Zelle `T[`i`]` mit Schlüssel x gefunden wird. Also aktualisiert **insert**(x, r') den richtigen Datensatz (und erzeugt kein Duplikat).

Zur **Korrektheit der Suche**: Aus der Invarianten folgt, dass **lookup**(x) nur dann einen Datensatz zurückgibt, wenn x vorher eingefügt wurde. Wenn dies der Fall ist, stellt die Invariante sicher, dass **lookup**(x) die eindeutig bestimmte Zelle `T[`i`]` mit Schlüssel x findet, und dass diese den aktuellen Datensatz für diesen Schlüssel enthält.

Erinnerung:

Erinnerung:

„Große“ Uniformitätsannahme (UF*):
Die Werte $h(x_1), \ldots, h(x_n)$ sind **rein zufällig**, d. h. uniform in $[m]$ und unabhängig.

Satz 5.4.1 (1962 bewiesen von Donald E. Knuth (*1938, TAOCP, TEX))

Satz 5.4.1 (1962 bewiesen von Donald E. Knuth (*1938, TAOCP, TEX))

Bei linearem Sondieren in einer Tabelle der Größe m mit Schlüsselmenge $S = \{x_1, \ldots, x_n\}$ gilt, mit festem $\boldsymbol{\alpha = \frac{n}{m}}$ (und $n, m \to \infty$):

Satz 5.4.1 (1962 bewiesen von Donald E. Knuth (*1938, TAOCP, TEX))

Bei linearem Sondieren in einer Tabelle der Größe m mit Schlüsselmenge $S = \{x_1, \ldots, x_n\}$ gilt, mit festem $\boldsymbol{\alpha = \frac{n}{m}}$ (und $n, m \to \infty$):

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

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

Satz 5.4.1 (1962 bewiesen von Donald E. Knuth (*1938, TAOCP, TEX))

Bei linearem Sondieren in einer Tabelle der Größe m mit Schlüsselmenge $S = \{x_1, \ldots, x_n\}$ gilt, mit festem $\boldsymbol{\alpha = \frac{n}{m}}$ (und $n, m \to \infty$):

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 $x_1, \ldots, x_n$) Anzahl von Schlüsselvergleichen bei **erfolgreicher Suche** nach $y \in S$ ist

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

Satz 5.4.1 (1962 bewiesen von Donald E. Knuth (*1938, TAOCP, TEX))

Bei linearem Sondieren in einer Tabelle der Größe m mit Schlüsselmenge $S = \{x_1, \ldots, x_n\}$ gilt, mit festem $\boldsymbol{\alpha = \frac{n}{m}}$ (und $n, m \to \infty$):

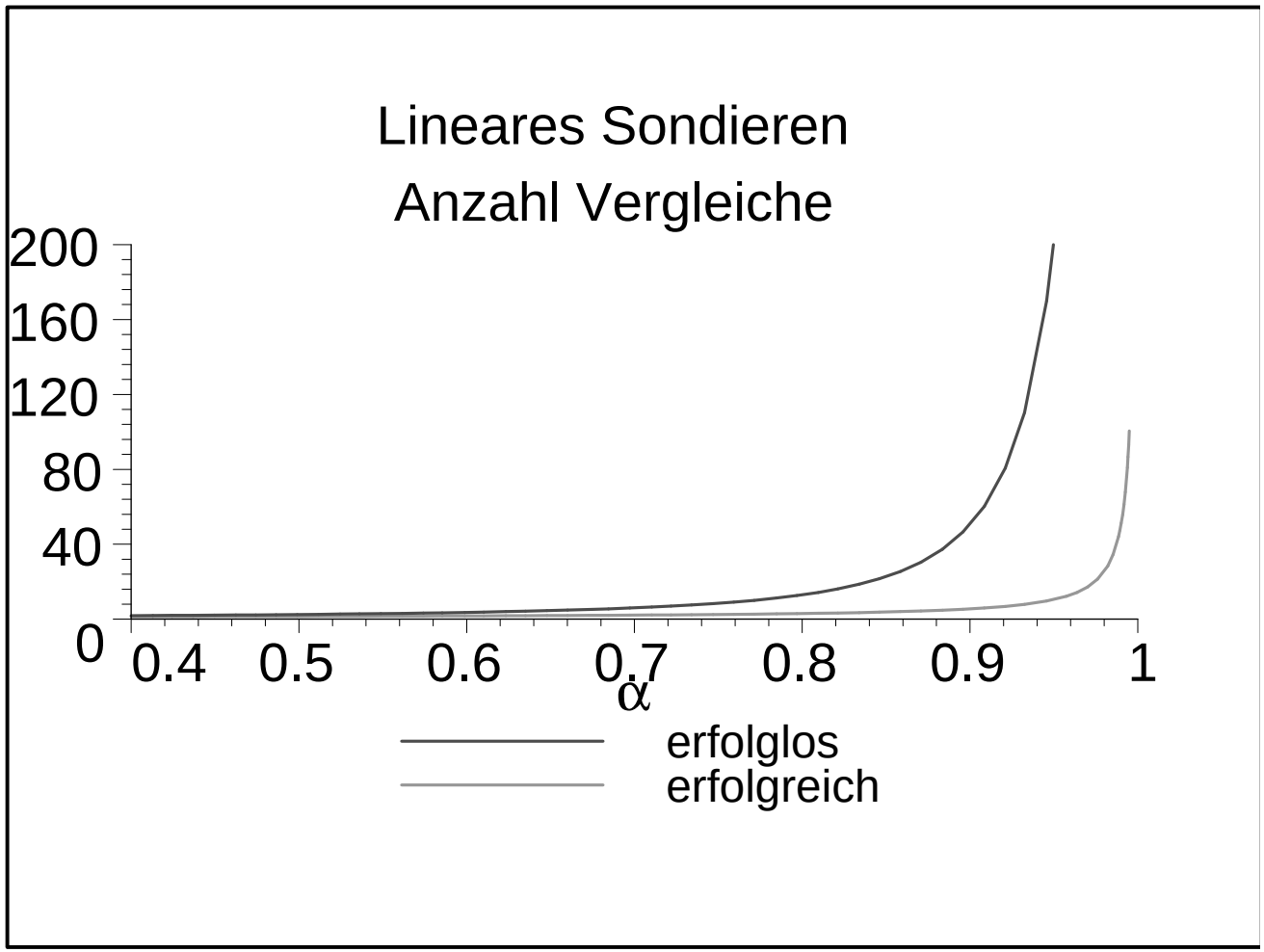
Die **erwartete** (nach (UF*)) Anzahl von Schlüsselvergleichen (besuchte Arrayzellen) bei **erfolgloser Suche** nach $y \notin S$ ist

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

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

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

Der erwartete Aufwand ist also $\boldsymbol{O(1)}$, falls $\boldsymbol{\alpha \leq \alpha_0}$ für eine Konstante $\alpha_0 < 1$.


Lineares Sondieren
Anzahl Vergleiche
200
160
120
80
40
0
0.4
0.5
0.6
0.7
0.8
0.9
1
α
erfolglos
erfolgreich

Erwartete Anzahl von Schlüsselvergleichen bei linearem Sondieren:

Erwartete Anzahl von Schlüsselvergleichen bei linearem Sondieren:

	erfolglose Suche	erfolgreiche Suche (Mittel über $x_1, \ldots, x_n$)
α	$\frac{1}{2} \cdot (1 + \frac{1}{(1-\alpha)^2})$	$\frac{1}{2} \cdot (1 + \frac{1}{1-\alpha})$

Erwartete Anzahl von Schlüsselvergleichen bei linearem Sondieren:

	erfolglose Suche	erfolgreiche Suche (Mittel über $x_1, \ldots, x_n$)
α	$\frac{1}{2} \cdot (1 + \frac{1}{(1-\alpha)^2})$	$\frac{1}{2} \cdot (1 + \frac{1}{1-\alpha})$
0,5	2,5	1,5

Erwartete Anzahl von Schlüsselvergleichen bei linearem Sondieren:

	erfolglose Suche	erfolgreiche Suche (Mittel über $x_1, \ldots, x_n$)
α	$\frac{1}{2} \cdot (1 + \frac{1}{(1-\alpha)^2})$	$\frac{1}{2} \cdot (1 + \frac{1}{1-\alpha})$
0,5	2,5	1,5
0,6	3,625	1,75

Erwartete Anzahl von Schlüsselvergleichen bei linearem Sondieren:

	erfolglose Suche	erfolgreiche Suche (Mittel über $x_1, \ldots, x_n$)
α	$\frac{1}{2} \cdot (1 + \frac{1}{(1-\alpha)^2})$	$\frac{1}{2} \cdot (1 + \frac{1}{1-\alpha})$
0,5	2,5	1,5
0,6	3,625	1,75
0,7	6,06	2,16

Erwartete Anzahl von Schlüsselvergleichen bei linearem Sondieren:

	erfolglose Suche	erfolgreiche Suche (Mittel über $x_1, \ldots, x_n$)
α	$\frac{1}{2} \cdot (1 + \frac{1}{(1-\alpha)^2})$	$\frac{1}{2} \cdot (1 + \frac{1}{1-\alpha})$
0,5	2,5	1,5
0,6	3,625	1,75
0,7	6,06	2,16
0,75	8,5	2,5

Erwartete Anzahl von Schlüsselvergleichen bei linearem Sondieren:

	erfolglose Suche	erfolgreiche Suche (Mittel über $x_1, \ldots, x_n$)
α	$\frac{1}{2} \cdot (1 + \frac{1}{(1-\alpha)^2})$	$\frac{1}{2} \cdot (1 + \frac{1}{1-\alpha})$
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

Erwartete Anzahl von Schlüsselvergleichen bei linearem Sondieren:

	erfolglose Suche	erfolgreiche Suche (Mittel über $x_1, \ldots, x_n$)
α	$\frac{1}{2} \cdot (1 + \frac{1}{(1-\alpha)^2})$	$\frac{1}{2} \cdot (1 + \frac{1}{1-\alpha})$
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**	**10,5**
0,98	**1250,5**	**25,5**

Ergebnisse für lineares Sondieren:

Ergebnisse für lineares Sondieren:

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

Ergebnisse für lineares Sondieren:

Eine einzelne erfolglose Suche wird z. B. bei zu 90% gefüllter Tabelle durchaus 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 durchaus 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 Suchkosten.

Ergebnisse für lineares Sondieren:

Eine einzelne erfolglose Suche wird z. B. bei zu 90% gefüllter Tabelle durchaus 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 Suchkosten.

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

Ergebnisse für lineares Sondieren:

Eine einzelne erfolglose Suche wird z. B. bei zu 90% gefüllter Tabelle durchaus 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 Suchkosten.

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

!!!

Ergebnisse für lineares Sondieren:

Eine einzelne erfolglose Suche wird z. B. bei zu 90% gefüllter Tabelle durchaus 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 Suchkosten.

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

Ergebnisse für lineares Sondieren:

Eine einzelne erfolglose Suche wird z. B. bei zu 90% gefüllter Tabelle durchaus 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 Suchkosten.

Die (über $x_i \in S$) **gemittelte** Einfüge-/Suchzeit ist dagegen sogar bei 90% gefüllter Tabelle 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!

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!

Alternative: **MurmurHash3** (32-Bit- oder 128-Bit-Werte).

`https://en.wikipedia.org/wiki/MurmurHash`

Lässt „seed“ zu, d. h. unterschiedliche Funktionen.

Bisherige experimentelle Untersuchungen: schnelle Auswertung, gute Verteilung.

Jedoch: Keine mathematische Aussage bekannt.

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!

Alternative: **MurmurHash3** (32-Bit- oder 128-Bit-Werte).

`https://en.wikipedia.org/wiki/MurmurHash`
Lässt „seed“ zu, d. h. unterschiedliche Funktionen.
Bisherige experimentelle Untersuchungen: schnelle Auswertung, gute Verteilung.
Jedoch: Keine mathematische Aussage bekannt.

Achtung! Kryptographische Hashfunktionen (z.B. MD4, MD5, SHA-1, SHA-3, . . .) haben andere Anforderungen und **viel höhere Auswertezeiten**!

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!

Alternative: **MurmurHash3** (32-Bit- oder 128-Bit-Werte).

`https://en.wikipedia.org/wiki/MurmurHash`
Lässt „seed“ zu, d. h. unterschiedliche Funktionen.
Bisherige experimentelle Untersuchungen: schnelle Auswertung, gute Verteilung.
Jedoch: Keine mathematische Aussage bekannt.

Achtung! Kryptographische Hashfunktionen (z.B. MD4, MD5, SHA-1, SHA-3, . . .) haben andere Anforderungen und **viel höhere Auswertezeiten**!
Sie sind für Anwendungen mit Hashtabellen nicht geeignet.

Andere Sondierungsfolgen sind möglich, zum Beispiel . . .

Andere Sondierungsfolgen sind möglich, zum Beispiel . . .

B – Quadratisches Sondieren

$h\colon U \to [m]$ sei beliebige Hashfunktion.

$$h(x,0) \quad = \quad h(x)$$

Andere Sondierungsfolgen sind möglich, zum Beispiel . . .

B – Quadratisches Sondieren

$h\colon U \to [m]$ sei beliebige Hashfunktion.

$$\begin{aligned} h(x,0) &= h(x) \\ h(x,1) &= (h(x)+1) \bmod m \end{aligned}$$

Andere Sondierungsfolgen sind möglich, zum Beispiel . . .

B – Quadratisches Sondieren

$h\colon U \to [m]$ sei beliebige Hashfunktion.

$$\begin{aligned} h(x,0) &= h(x) \\ h(x,1) &= (h(x)+1) \bmod m \\ h(x,2) &= (h(x)-1) \bmod m \end{aligned}$$

Andere Sondierungsfolgen sind möglich, zum Beispiel . . .

B – Quadratisches Sondieren

$h\colon U \to [m]$ sei beliebige Hashfunktion.

$$
\begin{aligned}
h(x,0) &= h(x) \\
h(x,1) &= (h(x) + \mathbf{1}) \bmod m \\
h(x,2) &= (h(x) - \mathbf{1}) \bmod m \\
h(x,3) &= (h(x) + \mathbf{4}) \bmod m
\end{aligned}
$$

Andere Sondierungsfolgen sind möglich, zum Beispiel . . .

B – Quadratisches Sondieren

$h\colon U \to [m]$ sei beliebige Hashfunktion.

$$
\begin{aligned}
h(x,0) &= h(x) \\
h(x,1) &= (h(x) + \mathbf{1}) \bmod m \\
h(x,2) &= (h(x) - \mathbf{1}) \bmod m \\
h(x,3) &= (h(x) + \mathbf{4}) \bmod m \\
h(x,4) &= (h(x) - \mathbf{4}) \bmod m
\end{aligned}
$$

Andere Sondierungsfolgen sind möglich, zum Beispiel . . .

B – Quadratisches Sondieren

$h\colon U \to [m]$ sei beliebige Hashfunktion.

$$
\begin{aligned}
h(x,0) &= h(x) \\
h(x,1) &= (h(x) + \mathbf{1}) \bmod m \\
h(x,2) &= (h(x) - \mathbf{1}) \bmod m \\
h(x,3) &= (h(x) + \mathbf{4}) \bmod m \\
h(x,4) &= (h(x) - \mathbf{4}) \bmod m \\
h(x,5) &= (h(x) + \mathbf{9}) \bmod m
\end{aligned}
$$

Andere Sondierungsfolgen sind möglich, zum Beispiel . . .

B – Quadratisches Sondieren

$h\colon U \to [m]$ sei beliebige Hashfunktion.

$$
\begin{aligned}
h(x,0) &= h(x) \\
h(x,1) &= (h(x) + \mathbf{1}) \bmod m \\
h(x,2) &= (h(x) - \mathbf{1}) \bmod m \\
h(x,3) &= (h(x) + \mathbf{4}) \bmod m \\
h(x,4) &= (h(x) - \mathbf{4}) \bmod m \\
h(x,5) &= (h(x) + \mathbf{9}) \bmod m \\
h(x,6) &= (h(x) - \mathbf{9}) \bmod m
\end{aligned}
$$

Andere Sondierungsfolgen sind möglich, zum Beispiel . . .

B – Quadratisches Sondieren

$h\colon U \to [m]$ sei beliebige Hashfunktion.

$$
\begin{aligned}
h(x,0) &= h(x) \\
h(x,1) &= (h(x) + \mathbf{1}) \bmod m \\
h(x,2) &= (h(x) - \mathbf{1}) \bmod m \\
h(x,3) &= (h(x) + \mathbf{4}) \bmod m \\
h(x,4) &= (h(x) - \mathbf{4}) \bmod m \\
h(x,5) &= (h(x) + \mathbf{9}) \bmod m \\
h(x,6) &= (h(x) - \mathbf{9}) \bmod m \\
&\vdots
\end{aligned}
$$

Allgemein: $\boxed{h(x,k) = \left(h(x) + \lceil k/2 \rceil^2 \cdot (-1)^{k+1}\right) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$

Andere Sondierungsfolgen sind möglich, zum Beispiel . . .

B – Quadratisches Sondieren

$h\colon U \to [m]$ sei beliebige Hashfunktion.

$$
\begin{aligned}
h(x,0) &= h(x) \\
h(x,1) &= (h(x) + \mathbf{1}) \bmod m \\
h(x,2) &= (h(x) - \mathbf{1}) \bmod m \\
h(x,3) &= (h(x) + \mathbf{4}) \bmod m \\
h(x,4) &= (h(x) - \mathbf{4}) \bmod m \\
h(x,5) &= (h(x) + \mathbf{9}) \bmod m \\
h(x,6) &= (h(x) - \mathbf{9}) \bmod m \\
&\vdots
\end{aligned}
$$

Allgemein: $\boxed{h(x,k) = \left(h(x) + \lceil k/2 \rceil^2 \cdot (-1)^{k+1}\right) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$

(Man kann die Folge auch inkrementell nur mit Additionen/Subtraktionen modulo m berechnen.)

Beispiel: $m = 19$,

Beispiel: $m = 19$, $h(x) = 8$,

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

8

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

8, **9**

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

$$\mathbf{8}, \mathbf{9}, \mathbf{7}$$

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

$$\mathbf{8}, \mathbf{9}, \mathbf{7}, \mathbf{12}$$

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

$$\mathbf{8}, \mathbf{9}, \mathbf{7}, \mathbf{12}, \mathbf{4}$$

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

$$\mathbf{8}, \mathbf{9}, \mathbf{7}, \mathbf{12}, \mathbf{4}, \mathbf{17}$$

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

$$\mathbf{8}, \mathbf{9}, \mathbf{7}, \mathbf{12}, \mathbf{4}, \mathbf{17}, \mathbf{18}$$

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

$$\mathbf{8}, \mathbf{9}, \mathbf{7}, \mathbf{12}, \mathbf{4}, \mathbf{17}, \mathbf{18}, \mathbf{5}$$

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

$$\mathbf{8}, \mathbf{9}, \mathbf{7}, \mathbf{12}, \mathbf{4}, \mathbf{17}, \mathbf{18}, \mathbf{5}, \mathbf{11}$$

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

$$\mathbf{8}, \mathbf{9}, \mathbf{7}, \mathbf{12}, \mathbf{4}, \mathbf{17}, \mathbf{18}, \mathbf{5}, \mathbf{11}, \mathbf{14}$$

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

$$\mathbf{8}, \mathbf{9}, \mathbf{7}, \mathbf{12}, \mathbf{4}, \mathbf{17}, \mathbf{18}, \mathbf{5}, \mathbf{11}, \mathbf{14}, \mathbf{2}$$

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

$$\mathbf{8}, \mathbf{9}, \mathbf{7}, \mathbf{12}, \mathbf{4}, \mathbf{17}, \mathbf{18}, \mathbf{5}, \mathbf{11}, \mathbf{14}, \mathbf{2}, \mathbf{6}$$

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

$$\mathbf{8}, \mathbf{9}, \mathbf{7}, \mathbf{12}, \mathbf{4}, \mathbf{17}, \mathbf{18}, \mathbf{5}, \mathbf{11}, \mathbf{14}, \mathbf{2}, \mathbf{6}, \mathbf{10}$$

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

$$\mathbf{8}, \mathbf{9}, \mathbf{7}, \mathbf{12}, \mathbf{4}, \mathbf{17}, \mathbf{18}, \mathbf{5}, \mathbf{11}, \mathbf{14}, \mathbf{2}, \mathbf{6}, \mathbf{10}, \mathbf{0}$$

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

$$\mathbf{8}, \mathbf{9}, \mathbf{7}, \mathbf{12}, \mathbf{4}, \mathbf{17}, \mathbf{18}, \mathbf{5}, \mathbf{11}, \mathbf{14}, \mathbf{2}, \mathbf{6}, \mathbf{10}, \mathbf{0}, \mathbf{16}$$

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

$$\mathbf{8}, \mathbf{9}, \mathbf{7}, \mathbf{12}, \mathbf{4}, \mathbf{17}, \mathbf{18}, \mathbf{5}, \mathbf{11}, \mathbf{14}, \mathbf{2}, \mathbf{6}, \mathbf{10}, \mathbf{0}, \mathbf{16}, \mathbf{15}$$

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

$$\mathbf{8}, \mathbf{9}, \mathbf{7}, \mathbf{12}, \mathbf{4}, \mathbf{17}, \mathbf{18}, \mathbf{5}, \mathbf{11}, \mathbf{14}, \mathbf{2}, \mathbf{6}, \mathbf{10}, \mathbf{0}, \mathbf{16}, \mathbf{15}, \mathbf{1}$$

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

$$\mathbf{8}, \mathbf{9}, \mathbf{7}, \mathbf{12}, \mathbf{4}, \mathbf{17}, \mathbf{18}, \mathbf{5}, \mathbf{11}, \mathbf{14}, \mathbf{2}, \mathbf{6}, \mathbf{10}, \mathbf{0}, \mathbf{16}, \mathbf{15}, \mathbf{1}, \mathbf{13}$$

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

$$\mathbf{8}, \mathbf{9}, \mathbf{7}, \mathbf{12}, \mathbf{4}, \mathbf{17}, \mathbf{18}, \mathbf{5}, \mathbf{11}, \mathbf{14}, \mathbf{2}, \mathbf{6}, \mathbf{10}, \mathbf{0}, \mathbf{16}, \mathbf{15}, \mathbf{1}, \mathbf{13}, \mathbf{3}.$$

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

$$8, 9, 7, 12, 4, 17, 18, 5, 11, 14, 2, 6, 10, 0, 16, 15, 1, 13, 3.$$

Fakt 5.4.2

Ist m Primzahl, so dass $m + 1$ durch 4 teilbar ist,

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

$$\mathbf{8}, \mathbf{9}, \mathbf{7}, \mathbf{12}, \mathbf{4}, \mathbf{17}, \mathbf{18}, \mathbf{5}, \mathbf{11}, \mathbf{14}, \mathbf{2}, \mathbf{6}, \mathbf{10}, \mathbf{0}, \mathbf{16}, \mathbf{15}, \mathbf{1}, \mathbf{13}, \mathbf{3}.$$

Fakt 5.4.2

Ist m Primzahl, so dass $m + 1$ durch 4 teilbar ist, dann gilt bei der Verwendung von quadratischem Sondieren:

$$(*) \qquad \{h(x, k) \mid 0 \leq k < m\} = [m].$$

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

$$\mathbf{8}, \mathbf{9}, \mathbf{7}, \mathbf{12}, \mathbf{4}, \mathbf{17}, \mathbf{18}, \mathbf{5}, \mathbf{11}, \mathbf{14}, \mathbf{2}, \mathbf{6}, \mathbf{10}, \mathbf{0}, \mathbf{16}, \mathbf{15}, \mathbf{1}, \mathbf{13}, \mathbf{3}.$$

Fakt 5.4.2

Ist m Primzahl, so dass $m + 1$ durch 4 teilbar ist, dann gilt bei der Verwendung von quadratischem Sondieren:

$$(*) \qquad \{h(x, k) \mid 0 \leq k < m\} = [m].$$

(*Beweis*: Elementare Zahlentheorie.)

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

$$\mathbf{8}, \mathbf{9}, \mathbf{7}, \mathbf{12}, \mathbf{4}, \mathbf{17}, \mathbf{18}, \mathbf{5}, \mathbf{11}, \mathbf{14}, \mathbf{2}, \mathbf{6}, \mathbf{10}, \mathbf{0}, \mathbf{16}, \mathbf{15}, \mathbf{1}, \mathbf{13}, \mathbf{3}.$$

Fakt 5.4.2

Ist m Primzahl, so dass $m + 1$ durch 4 teilbar ist, dann gilt bei der Verwendung von quadratischem Sondieren:

$$(*) \qquad \{h(x, k) \mid 0 \leq k < m\} = [m].$$

(*Beweis*: Elementare Zahlentheorie.)

Beobachtung (empirisch):

Quadratisches Sondieren verhält sich sehr gut.

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

$$\mathbf{8}, \mathbf{9}, \mathbf{7}, \mathbf{12}, \mathbf{4}, \mathbf{17}, \mathbf{18}, \mathbf{5}, \mathbf{11}, \mathbf{14}, \mathbf{2}, \mathbf{6}, \mathbf{10}, \mathbf{0}, \mathbf{16}, \mathbf{15}, \mathbf{1}, \mathbf{13}, \mathbf{3}.$$

Fakt 5.4.2

Ist m Primzahl, so dass $m + 1$ durch 4 teilbar ist, dann gilt bei der Verwendung von quadratischem Sondieren:

$$(*) \qquad \{h(x, k) \mid 0 \leq k < m\} = [m].$$

(*Beweis*: Elementare Zahlentheorie.)

Beobachtung (empirisch):

Quadratisches Sondieren verhält sich sehr gut.

(**Vor.:** h gut verteilend, Auslastungsfaktor $\alpha \leq 0{,}9$.)

C – Doppel-Hashing

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 : U \to [m]$$
$$h_2 : U \to [m-1]$$

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 : U \to [m]$$
$$h_2 : U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 : U \to [m]$$
$$h_2 : U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

$$\boxed{h(x, \boldsymbol{k}) := (h_1(x) + \boldsymbol{k} \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$$

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 : U \to [m]$$
$$h_2 : U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

$$\boxed{h(x, \boldsymbol{k}) := (h_1(x) + \boldsymbol{k} \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$$

Beispiel: $m = 13$,

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 \colon U \to [m]$$
$$h_2 \colon U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

$$\boxed{h(x, \boldsymbol{k}) := (h_1(x) + \boldsymbol{k} \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$$

Beispiel: $m = 13$, $h_1(x) = \mathbf{3}$,

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 \colon U \to [m]$$
$$h_2 \colon U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

$$\boxed{h(x, \boldsymbol{k}) := (h_1(x) + \boldsymbol{k} \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$$

Beispiel: $m = 13$, $h_1(x) = \mathbf{3}$, $1 + h_2(x) = \mathbf{7}$

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 \colon U \to [m]$$
$$h_2 \colon U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

$$\boxed{h(x, \boldsymbol{k}) := (h_1(x) + \boldsymbol{k} \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$$

Beispiel: $m = 13$, $h_1(x) = \mathbf{3}$, $1 + h_2(x) = \mathbf{7}$ liefert Sondierungsfolge

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 \colon U \to [m]$$
$$h_2 \colon U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

$$\boxed{h(x, \boldsymbol{k}) := (h_1(x) + \boldsymbol{k} \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$$

Beispiel: $m = 13$, $h_1(x) = \mathbf{3}$, $1 + h_2(x) = \mathbf{7}$ liefert Sondierungsfolge

$\mathbf{3}$

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 \colon U \to [m]$$
$$h_2 \colon U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

$$\boxed{h(x, \boldsymbol{k}) := (h_1(x) + \boldsymbol{k} \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$$

Beispiel: $m = 13$, $h_1(x) = \mathbf{3}$, $1 + h_2(x) = \mathbf{7}$ liefert Sondierungsfolge

$$\mathbf{3}, \mathbf{10}$$

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 \colon U \to [m]$$
$$h_2 \colon U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

$$\boxed{h(x, \boldsymbol{k}) := (h_1(x) + \boldsymbol{k} \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$$

Beispiel: $m = 13$, $h_1(x) = \mathbf{3}$, $1 + h_2(x) = \mathbf{7}$ liefert Sondierungsfolge

$$\mathbf{3}, \mathbf{10}, \mathbf{4}$$

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 \colon U \to [m]$$
$$h_2 \colon U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

$$\boxed{h(x, \boldsymbol{k}) := (h_1(x) + \boldsymbol{k} \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$$

Beispiel: $m = 13$, $h_1(x) = \mathbf{3}$, $1 + h_2(x) = \mathbf{7}$ liefert Sondierungsfolge

$$\mathbf{3}, \mathbf{10}, \mathbf{4}, \mathbf{11}$$

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 \colon U \to [m]$$
$$h_2 \colon U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

$$\boxed{h(x, \boldsymbol{k}) := (h_1(x) + \boldsymbol{k} \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$$

Beispiel: $m = 13$, $h_1(x) = \mathbf{3}$, $1 + h_2(x) = \mathbf{7}$ liefert Sondierungsfolge

$$\mathbf{3}, \mathbf{10}, \mathbf{4}, \mathbf{11}, \mathbf{5}$$

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 \colon U \to [m]$$
$$h_2 \colon U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

$$\boxed{h(x, \boldsymbol{k}) := (h_1(x) + \boldsymbol{k} \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$$

Beispiel: $m = 13$, $h_1(x) = \mathbf{3}$, $1 + h_2(x) = \mathbf{7}$ liefert Sondierungsfolge

$$\mathbf{3}, \mathbf{10}, \mathbf{4}, \mathbf{11}, \mathbf{5}, \mathbf{12}$$

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 \colon U \to [m]$$
$$h_2 \colon U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

$$\boxed{h(x, \boldsymbol{k}) := (h_1(x) + \boldsymbol{k} \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$$

Beispiel: $m = 13$, $h_1(x) = \mathbf{3}$, $1 + h_2(x) = \mathbf{7}$ liefert Sondierungsfolge

$$\mathbf{3}, \mathbf{10}, \mathbf{4}, \mathbf{11}, \mathbf{5}, \mathbf{12}, \mathbf{6}$$

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 \colon U \to [m]$$
$$h_2 \colon U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

$$\boxed{h(x, \boldsymbol{k}) := (h_1(x) + \boldsymbol{k} \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$$

Beispiel: $m = 13$, $h_1(x) = \mathbf{3}$, $1 + h_2(x) = \mathbf{7}$ liefert Sondierungsfolge

$$\mathbf{3}, \mathbf{10}, \mathbf{4}, \mathbf{11}, \mathbf{5}, \mathbf{12}, \mathbf{6}, \mathbf{0}$$

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 \colon U \to [m]$$
$$h_2 \colon U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

$$\boxed{h(x, \boldsymbol{k}) := (h_1(x) + \boldsymbol{k} \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$$

Beispiel: $m = 13$, $h_1(x) = \mathbf{3}$, $1 + h_2(x) = \mathbf{7}$ liefert Sondierungsfolge

$$\mathbf{3}, \mathbf{10}, \mathbf{4}, \mathbf{11}, \mathbf{5}, \mathbf{12}, \mathbf{6}, \mathbf{0}, \mathbf{7}$$

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 \colon U \to [m]$$
$$h_2 \colon U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

$$\boxed{h(x, \boldsymbol{k}) := (h_1(x) + \boldsymbol{k} \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$$

Beispiel: $m = 13$, $h_1(x) = \mathbf{3}$, $1 + h_2(x) = \mathbf{7}$ liefert Sondierungsfolge

$$\mathbf{3}, \mathbf{10}, \mathbf{4}, \mathbf{11}, \mathbf{5}, \mathbf{12}, \mathbf{6}, \mathbf{0}, \mathbf{7}, \mathbf{1}$$

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 \colon U \to [m]$$
$$h_2 \colon U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

$$\boxed{h(x, \boldsymbol{k}) := (h_1(x) + \boldsymbol{k} \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$$

Beispiel: $m = 13$, $h_1(x) = \mathbf{3}$, $1 + h_2(x) = \mathbf{7}$ liefert Sondierungsfolge

$$\mathbf{3}, \mathbf{10}, \mathbf{4}, \mathbf{11}, \mathbf{5}, \mathbf{12}, \mathbf{6}, \mathbf{0}, \mathbf{7}, \mathbf{1}, \mathbf{8}$$

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 \colon U \to [m]$$
$$h_2 \colon U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

$$\boxed{h(x, \boldsymbol{k}) := (h_1(x) + \boldsymbol{k} \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$$

Beispiel: $m = 13$, $h_1(x) = \mathbf{3}$, $1 + h_2(x) = \mathbf{7}$ liefert Sondierungsfolge

$$\mathbf{3}, \mathbf{10}, \mathbf{4}, \mathbf{11}, \mathbf{5}, \mathbf{12}, \mathbf{6}, \mathbf{0}, \mathbf{7}, \mathbf{1}, \mathbf{8}, \mathbf{2}$$

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 \colon U \to [m]$$
$$h_2 \colon U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

$$\boxed{h(x, \boldsymbol{k}) := (h_1(x) + \boldsymbol{k} \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$$

Beispiel: $m = 13$, $h_1(x) = \mathbf{3}$, $1 + h_2(x) = \mathbf{7}$ liefert Sondierungsfolge

$$\mathbf{3}, \mathbf{10}, \mathbf{4}, \mathbf{11}, \mathbf{5}, \mathbf{12}, \mathbf{6}, \mathbf{0}, \mathbf{7}, \mathbf{1}, \mathbf{8}, \mathbf{2}, \mathbf{9}.$$

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 \colon U \to [m]$$
$$h_2 \colon U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

$$\boxed{h(x, \boldsymbol{k}) := (h_1(x) + \boldsymbol{k} \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$$

Beispiel: $m = 13$, $h_1(x) = \mathbf{3}$, $1 + h_2(x) = \mathbf{7}$ liefert Sondierungsfolge

$$\mathbf{3}, \mathbf{10}, \mathbf{4}, \mathbf{11}, \mathbf{5}, \mathbf{12}, \mathbf{6}, \mathbf{0}, \mathbf{7}, \mathbf{1}, \mathbf{8}, \mathbf{2}, \mathbf{9}.$$

Übung: Berechne die Folgenglieder **inkrementell** (Runden $k = 0, 1, 2, \ldots$) ohne Multiplikationen und Divisionen („$\ldots \bmod m$“). *Beispiel*: $\mathbf{3}$

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 \colon U \to [m]$$
$$h_2 \colon U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

$$\boxed{h(x, \boldsymbol{k}) := (h_1(x) + \boldsymbol{k} \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$$

Beispiel: $m = 13$, $h_1(x) = \mathbf{3}$, $1 + h_2(x) = \mathbf{7}$ liefert Sondierungsfolge

$$\mathbf{3}, \mathbf{10}, \mathbf{4}, \mathbf{11}, \mathbf{5}, \mathbf{12}, \mathbf{6}, \mathbf{0}, \mathbf{7}, \mathbf{1}, \mathbf{8}, \mathbf{2}, \mathbf{9}.$$

Übung: Berechne die Folgenglieder **inkrementell** (Runden $k = 0, 1, 2, \ldots$) ohne Multiplikationen und Divisionen („$\ldots \bmod m$“). *Beispiel*: $\mathbf{3}, 3 + \mathbf{7} = 10$

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 \colon U \to [m]$$
$$h_2 \colon U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

$$\boxed{h(x, \boldsymbol{k}) := (h_1(x) + \boldsymbol{k} \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$$

Beispiel: $m = 13$, $h_1(x) = \mathbf{3}$, $1 + h_2(x) = \mathbf{7}$ liefert Sondierungsfolge

$$\mathbf{3}, \mathbf{10}, \mathbf{4}, \mathbf{11}, \mathbf{5}, \mathbf{12}, \mathbf{6}, \mathbf{0}, \mathbf{7}, \mathbf{1}, \mathbf{8}, \mathbf{2}, \mathbf{9}.$$

Übung: Berechne die Folgenglieder **inkrementell** (Runden $k = 0, 1, 2, \ldots$) ohne Multiplikationen und Divisionen („$\ldots \bmod m$“). *Beispiel*: $\mathbf{3}, 3 + \mathbf{7} = 10, (10 + \mathbf{7}) \bmod 13 = 4$

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 \colon U \to [m]$$
$$h_2 \colon U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

$$\boxed{h(x, \boldsymbol{k}) := (h_1(x) + \boldsymbol{k} \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$$

Beispiel: $m = 13$, $h_1(x) = \mathbf{3}$, $1 + h_2(x) = \mathbf{7}$ liefert Sondierungsfolge

$$\mathbf{3}, \mathbf{10}, \mathbf{4}, \mathbf{11}, \mathbf{5}, \mathbf{12}, \mathbf{6}, \mathbf{0}, \mathbf{7}, \mathbf{1}, \mathbf{8}, \mathbf{2}, \mathbf{9}.$$

Übung: Berechne die Folgenglieder **inkrementell** (Runden $k = 0, 1, 2, \ldots$) ohne Multiplikationen und Divisionen („$\ldots \bmod m$“). *Beispiel*: $\mathbf{3}, 3 + \mathbf{7} = 10, (10 + \mathbf{7}) \bmod 13 = 4, \ldots$

$h(x,k) := (h_1(x) + k \cdot (1 + h_2(x))) \bmod m$, für $k = 0, 1, 2, \ldots$.

$h(x,k) := (h_1(x) + k \cdot (1 + h_2(x))) \bmod m$, für $k = 0, 1, 2, \ldots$.

Proposition 5.4.3

Wenn m Primzahl ist, dann gilt für **Doppel-Hashing**:

$h(x,k) := (h_1(x) + k \cdot (1 + h_2(x))) \bmod m$, für $k = 0, 1, 2, \ldots$.

Proposition 5.4.3

Wenn m Primzahl ist, dann gilt für **Doppel-Hashing**:

(∗) $\{h(x,k) \mid 0 \leq k < m\} = [m]$.

$h(x,k) := (h_1(x) + k \cdot (1 + h_2(x))) \bmod m$, für $k = 0, 1, 2, \ldots$.

Proposition 5.4.3

Wenn m Primzahl ist, dann gilt für **Doppel-Hashing**:

(∗) $\{h(x,k) \mid 0 \leq k < m\} = [m]$.

Beweis: Einfache Zahlentheorie, nicht prüfungsrelevant. (Siehe Druckfolien.)

$h(x,k) := (h_1(x) + k \cdot (1 + h_2(x))) \bmod m$, für $k = 0, 1, 2, \ldots$.

Proposition 5.4.3

Wenn m Primzahl ist, dann gilt für **Doppel-Hashing**:

(*) $\{h(x,k) \mid 0 \leq k < m\} = [m]$.

Beweis: Einfache Zahlentheorie, nicht prüfungsrelevant. (Siehe Druckfolien.)

Man kann beweisen: Wenn $h_1(x), h_2(x)$, $x \in S$, rein zufällig sind, verhält sich **Doppel-Hashing** sehr gut, nämlich im Wesentlichen wie „Uniformes Sondieren“.

Vorlesungsvideo:

Uniformes Sondieren

D – Uniformes Sondieren/Ideales Hashing

D – Uniformes Sondieren/Ideales Hashing

Keine Methode, sondern eine **Wahrscheinlichkeitsannahme**
für Verfahren mit offener Adressierung:

D – Uniformes Sondieren/Ideales Hashing

Keine Methode, sondern eine **Wahrscheinlichkeitsannahme** für Verfahren mit offener Adressierung:

(IH): $(h(x,0), h(x,1), \ldots, h(x,m-1))$ ist eine **rein zufällige** Permutation von $[m]$, unabhängig für jedes $x \in U$.

D – Uniformes Sondieren/Ideales Hashing

Keine Methode, sondern eine **Wahrscheinlichkeitsannahme** für Verfahren mit offener Adressierung:

(IH):

$(h(x,0), h(x,1), \ldots, h(x,m-1))$ ist eine **rein zufällige** Permutation von $[m]$, unabhängig für jedes $x \in U$.

(Erinnerung: Es gibt $m!$ verschiedene Permutationen von $[m]$.)

D – Uniformes Sondieren/Ideales Hashing

Keine Methode, sondern eine **Wahrscheinlichkeitsannahme** für Verfahren mit offener Adressierung:

(IH): $(h(x,0), h(x,1), \ldots, h(x,m-1))$ ist eine **rein zufällige** Permutation von $[m]$, unabhängig für jedes $x \in U$.

(Erinnerung: Es gibt $m!$ verschiedene Permutationen von $[m]$.)

Effekt von (IH): $x_1, \ldots, x_n$ seien gespeichert; y kommt neu hinzu.

D – Uniformes Sondieren/Ideales Hashing

Keine Methode, sondern eine **Wahrscheinlichkeitsannahme** für Verfahren mit offener Adressierung:

(IH): $(h(x,0), h(x,1), \ldots, h(x,m-1))$ ist eine **rein zufällige** Permutation von $[m]$, unabhängig für jedes $x \in U$.

(Erinnerung: Es gibt $m!$ verschiedene Permutationen von $[m]$.)

Effekt von (IH): $x_1, \ldots, x_n$ seien gespeichert; y kommt neu hinzu.
Ganz egal was die Hashwerte von $x_1, \ldots, x_n$ und was $h(y,0), \ldots, h(y,k-1)$ sind:

D – Uniformes Sondieren/Ideales Hashing

Keine Methode, sondern eine **Wahrscheinlichkeitsannahme** für Verfahren mit offener Adressierung:

(IH): $(h(x,0), h(x,1), \ldots, h(x,m-1))$ ist eine **rein zufällige** Permutation von $[m]$, unabhängig für jedes $x \in U$.

(Erinnerung: Es gibt $m!$ verschiedene Permutationen von $[m]$.)

Effekt von (IH): $x_1, \ldots, x_n$ seien gespeichert; y kommt neu hinzu.
Ganz egal was die Hashwerte von $x_1, \ldots, x_n$ und was $h(y,0), \ldots, h(y,k-1)$ sind:
$h(y,k)$

D – Uniformes Sondieren/Ideales Hashing

Keine Methode, sondern eine **Wahrscheinlichkeitsannahme** für Verfahren mit offener Adressierung:

(IH): $(h(x,0), h(x,1), \ldots, h(x,m-1))$ ist eine **rein zufällige** Permutation von $[m]$, unabhängig für jedes $x \in U$.

(Erinnerung: Es gibt $m!$ verschiedene Permutationen von $[m]$.)

Effekt von (IH): $x_1, \ldots, x_n$ seien gespeichert; y kommt neu hinzu.
Ganz egal was die Hashwerte von $x_1, \ldots, x_n$ und was $h(y,0), \ldots, h(y,k-1)$ sind:
$h(y,k)$ nimmt jeden Wert $j \in [m] - \{h(y,0), \ldots, h(y,k-1)\}$

D – Uniformes Sondieren/Ideales Hashing

Keine Methode, sondern eine **Wahrscheinlichkeitsannahme** für Verfahren mit offener Adressierung:

(IH): $(h(x,0), h(x,1), \ldots, h(x,m-1))$ ist eine **rein zufällige** Permutation von $[m]$, unabhängig für jedes $x \in U$.

(Erinnerung: Es gibt $m!$ verschiedene Permutationen von $[m]$.)

Effekt von (IH): $x_1, \ldots, x_n$ seien gespeichert; y kommt neu hinzu.
Ganz egal was die Hashwerte von $x_1, \ldots, x_n$ und was $h(y,0), \ldots, h(y,k-1)$ sind:
$h(y,k)$ nimmt jeden Wert $j \in [m] - \{h(y,0), \ldots, h(y,k-1)\}$
mit derselben Wahrscheinlichkeit $1/(m-k)$ an.

$\boldsymbol{n}$: Anzahl der Schlüssel in $\mathtt{T}[0\,..\,m-1]$;

$\boldsymbol{n}$: Anzahl der Schlüssel in $\mathtt{T}[0\,..\,m-1]$; $\boldsymbol{\alpha} = \frac{n}{m}$.

$\boldsymbol{n}$: Anzahl der Schlüssel in $\texttt{T}[0\,..\,m-1]$; $\boldsymbol{\alpha} = \frac{n}{m}$.

$\boldsymbol{T'_{m,n}}$:= #(untersuchte Fächer bei **erfolgloser Suche**) (eine Zufallsvariable)

$\boldsymbol{n}$: Anzahl der Schlüssel in $\texttt{T}[0\,..\,m-1]$; $\boldsymbol{\alpha} = \frac{n}{m}$.

$\boldsymbol{T'_{m,n}}$:= #(untersuchte Fächer bei **erfolgloser Suche**) (eine Zufallsvariable)

$\boldsymbol{C'_{m,n}} := \mathbf{E}(T'_{m,n})$.

$\boldsymbol{n}$: Anzahl der Schlüssel in $\texttt{T}[0\,..\,m-1]$; $\boldsymbol{\alpha} = \frac{n}{m}$.

$\boldsymbol{T'_{m,n}}$:= #(untersuchte Fächer bei **erfolgloser Suche**) (eine Zufallsvariable)

$\boldsymbol{C'_{m,n}} := \mathbf{E}(T'_{m,n})$.

$\boldsymbol{T_{m,n}}$:= #(untersuchte Fächer bei **erfolgreicher Suche**), gemittelt über $x_1, \ldots, x_n$ (eine Zufallsvariable).

$\boldsymbol{n}$: Anzahl der Schlüssel in $\texttt{T}[0\,..\,m-1]$; $\boldsymbol{\alpha} = \frac{n}{m}$.

$\boldsymbol{T'_{m,n}}$:= #(untersuchte Fächer bei **erfolgloser Suche**) (eine Zufallsvariable)

$\boldsymbol{C'_{m,n}} := \mathbf{E}(T'_{m,n})$.

$\boldsymbol{T_{m,n}}$:= #(untersuchte Fächer bei **erfolgreicher Suche**), gemittelt über $x_1, \ldots, x_n$ (eine Zufallsvariable).

$\boldsymbol{C_{m,n}} := \mathbf{E}(T_{m,n})$.

$\boldsymbol{n}$: Anzahl der Schlüssel in $\texttt{T}[0\,..\,m-1]$; $\boldsymbol{\alpha} = \frac{n}{m}$.

$\boldsymbol{T'_{m,n}} := \#$(untersuchte Fächer bei **erfolgloser Suche**) (eine Zufallsvariable)

$\boldsymbol{C'_{m,n}} := \mathbf{E}(T'_{m,n})$.

$\boldsymbol{T_{m,n}} := \#$(untersuchte Fächer bei **erfolgreicher Suche**), gemittelt über $x_1, \ldots, x_n$ (eine Zufallsvariable).

$\boldsymbol{C_{m,n}} := \mathbf{E}(T_{m,n})$.

Satz 5.4.5

$\boldsymbol{n}$: Anzahl der Schlüssel in $\texttt{T}[0\,..\,m-1]$; $\boldsymbol{\alpha} = \frac{n}{m}$.

$\boldsymbol{T'_{m,n}}$:= #(untersuchte Fächer bei **erfolgloser Suche**) (eine Zufallsvariable)

$\boldsymbol{C'_{m,n}} := \mathbf{E}(T'_{m,n})$.

$\boldsymbol{T_{m,n}}$:= #(untersuchte Fächer bei **erfolgreicher Suche**), gemittelt über $x_1, \ldots, x_n$ (eine Zufallsvariable).

$\boldsymbol{C_{m,n}} := \mathbf{E}(T_{m,n})$.

Satz 5.4.5

Unter der Annahme (IH) gilt, für $\alpha = n/m$:

$\boldsymbol{n}$: Anzahl der Schlüssel in $\mathtt{T}[0\,..\,m-1]$; $\boldsymbol{\alpha} = \frac{n}{m}$.

$\boldsymbol{T'_{m,n}}$:= #(untersuchte Fächer bei **erfolgloser Suche**) (eine Zufallsvariable)

$\boldsymbol{C'_{m,n}} := \mathbf{E}(T'_{m,n})$.

$\boldsymbol{T_{m,n}}$:= #(untersuchte Fächer bei **erfolgreicher Suche**), gemittelt über $x_1, \ldots, x_n$ (eine Zufallsvariable).

$\boldsymbol{C_{m,n}} := \mathbf{E}(T_{m,n})$.

Satz 5.4.5

Unter der Annahme (IH) gilt, für $\alpha = n/m$:

(i) $\boldsymbol{C'_{m,n}}$

$\boldsymbol{n}$: Anzahl der Schlüssel in $\mathtt{T}[0\,..\,m-1]$; $\boldsymbol{\alpha} = \frac{n}{m}$.

$\boldsymbol{T'_{m,n}}$:= #(untersuchte Fächer bei **erfolgloser Suche**) (eine Zufallsvariable)

$\boldsymbol{C'_{m,n}} := \mathbf{E}(T'_{m,n})$.

$\boldsymbol{T_{m,n}}$:= #(untersuchte Fächer bei **erfolgreicher Suche**), gemittelt über $x_1, \ldots, x_n$ (eine Zufallsvariable).

$\boldsymbol{C_{m,n}} := \mathbf{E}(T_{m,n})$.

Satz 5.4.5

Unter der Annahme (IH) gilt, für $\alpha = n/m$:

(i) $\boldsymbol{C'_{m,n}} = \frac{m+1}{m+1-n}$

n: Anzahl der Schlüssel in $\mathtt{T}[0\,..\,m-1]$; $\alpha = \frac{n}{m}$.

$T'_{m,n}$:= #(untersuchte Fächer bei **erfolgloser Suche**) (eine Zufallsvariable)

$C'_{m,n} := \mathbf{E}(T'_{m,n})$.

$T_{m,n}$:= #(untersuchte Fächer bei **erfolgreicher Suche**), gemittelt über $x_1, \ldots, x_n$ (eine Zufallsvariable).

$C_{m,n} := \mathbf{E}(T_{m,n})$.

Satz 5.4.5

Unter der Annahme (IH) gilt, für $\alpha = n/m$:

(i) $C'_{m,n} = \frac{m+1}{m+1-n} = \frac{1}{1-\frac{n}{m+1}} \quad \left(< \frac{1}{1-n/m} = \frac{1}{1-\alpha} \right)$.

n: Anzahl der Schlüssel in $\mathtt{T}[0\,..\,m-1]$; $\alpha = \frac{n}{m}$.

$T'_{m,n}$:= #(untersuchte Fächer bei **erfolgloser Suche**) (eine Zufallsvariable)

$C'_{m,n} := \mathbf{E}(T'_{m,n})$.

$T_{m,n}$:= #(untersuchte Fächer bei **erfolgreicher Suche**), gemittelt über $x_1, \ldots, x_n$ (eine Zufallsvariable).

$C_{m,n} := \mathbf{E}(T_{m,n})$.

Satz 5.4.5

Unter der Annahme (IH) gilt, für $\alpha = n/m$:

(i) $C'_{m,n} = \frac{m+1}{m+1-n} = \frac{1}{1-\frac{n}{m+1}} \quad \left(< \frac{1}{1-n/m} = \frac{1}{1-\alpha} \right)$.

Wenn α fest und $n, m \to \infty$, dann gilt $C'_{m,n} \to \frac{1}{1-\alpha}$.

n: Anzahl der Schlüssel in $\mathtt{T}[0\,..\,m-1]$; $\alpha = \frac{n}{m}$.

$T'_{m,n}$:= #(untersuchte Fächer bei **erfolgloser Suche**) (eine Zufallsvariable)

$C'_{m,n} := \mathbf{E}(T'_{m,n})$.

$T_{m,n}$:= #(untersuchte Fächer bei **erfolgreicher Suche**), gemittelt über $x_1, \ldots, x_n$ (eine Zufallsvariable).

$C_{m,n} := \mathbf{E}(T_{m,n})$.

Satz 5.4.5

Unter der Annahme (IH) gilt, für $\alpha = n/m$:

(i) $C'_{m,n} = \frac{m+1}{m+1-n} = \frac{1}{1-\frac{n}{m+1}} \quad \left(< \frac{1}{1-n/m} = \frac{1}{1-\alpha} \right)$.

Wenn α fest und $n, m \to \infty$, dann gilt $C'_{m,n} \to \frac{1}{1-\alpha}$.

(Immer wieder **rein zufällige** Positionen zu testen liefert Erfolgswahrscheinlichkeit $\frac{m-n}{m}$ in einem Versuch, also erwartete Versuchsanzahl $\frac{m}{m-n} = \frac{1}{1-\alpha}$.)

Beweis von (i): $x_1, \ldots, x_n$ sind gespeichert, $y \notin \{x_1, \ldots, x_n\}$ wird gesucht.

Beweis von (i): $x_1, \ldots, x_n$ sind gespeichert, $y \notin \{x_1, \ldots, x_n\}$ wird gesucht.

In jedem Fall wird $\texttt{T}[h(y, 0)]$ untersucht, Kosten 1.

Beweis von (i): $x_1, \ldots, x_n$ sind gespeichert, $y \notin \{x_1, \ldots, x_n\}$ wird gesucht.

In jedem Fall wird $\texttt{T}[h(y,0)]$ untersucht, Kosten 1.

1. *Fall*: $\texttt{T}[h(y,0)]$ ist leer.

Beweis von (i): $x_1, \ldots, x_n$ sind gespeichert, $y \notin \{x_1, \ldots, x_n\}$ wird gesucht.

In jedem Fall wird $\texttt{T}[h(y,0)]$ untersucht, Kosten 1.

1. *Fall*: $\texttt{T}[h(y,0)]$ ist leer.

Wahrscheinlichkeit: $\frac{m-n}{m}$.

Beweis von (i): $x_1, \ldots, x_n$ sind gespeichert, $y \notin \{x_1, \ldots, x_n\}$ wird gesucht.

In jedem Fall wird $\texttt{T}[h(y,0)]$ untersucht, Kosten 1.

1. *Fall*: $\texttt{T}[h(y,0)]$ ist leer.

Wahrscheinlichkeit: $\frac{m-n}{m}$. Weitere Kosten: 0.

Beweis von (i): $x_1, \ldots, x_n$ sind gespeichert, $y \notin \{x_1, \ldots, x_n\}$ wird gesucht.

In jedem Fall wird $\texttt{T}[h(y,0)]$ untersucht, Kosten 1.

1. *Fall*: $\texttt{T}[h(y,0)]$ ist leer.

Wahrscheinlichkeit: $\frac{m-n}{m}$. Weitere Kosten: 0.

2. *Fall*: $\texttt{T}[h(y,0)]$ ist **nicht leer**.

Beweis von (i): $x_1, \ldots, x_n$ sind gespeichert, $y \notin \{x_1, \ldots, x_n\}$ wird gesucht.

In jedem Fall wird $\texttt{T}[h(y,0)]$ untersucht, Kosten 1.

1. *Fall*: $\texttt{T}[h(y,0)]$ ist leer.

Wahrscheinlichkeit: $\frac{m-n}{m}$. Weitere Kosten: 0.

2. *Fall*: $\texttt{T}[h(y,0)]$ ist **nicht leer**.

Wahrscheinlichkeit: $\frac{n}{m}$.

Beweis von (i): $x_1, \ldots, x_n$ sind gespeichert, $y \notin \{x_1, \ldots, x_n\}$ wird gesucht.

In jedem Fall wird $\texttt{T}[h(y,0)]$ untersucht, Kosten 1.

1. *Fall*: $\texttt{T}[h(y,0)]$ ist leer.

Wahrscheinlichkeit: $\frac{m-n}{m}$. Weitere Kosten: 0.

2. *Fall*: $\texttt{T}[h(y,0)]$ ist **nicht leer**.

Wahrscheinlichkeit: $\frac{n}{m}$. Weitere Kosten: $\boldsymbol{C'_{m-1,n-1}}$.

Beweis von (i): $x_1, \ldots, x_n$ sind gespeichert, $y \notin \{x_1, \ldots, x_n\}$ wird gesucht.

In jedem Fall wird $\texttt{T}[h(y,0)]$ untersucht, Kosten 1.

1. *Fall*: $\texttt{T}[h(y,0)]$ ist leer.

Wahrscheinlichkeit: $\frac{m-n}{m}$. Weitere Kosten: 0.

2. *Fall*: $\texttt{T}[h(y,0)]$ ist **nicht leer**.

Wahrscheinlichkeit: $\frac{n}{m}$. Weitere Kosten: $\boldsymbol{C'_{m-1,n-1}}$.

(Suche in $m-1$ Plätzen, von denen $n-1$ besetzt sind.)

Beweis von (i): $x_1, \ldots, x_n$ sind gespeichert, $y \notin \{x_1, \ldots, x_n\}$ wird gesucht.

In jedem Fall wird $\texttt{T}[h(y,0)]$ untersucht, Kosten 1.

1. *Fall*: $\texttt{T}[h(y,0)]$ ist leer.

Wahrscheinlichkeit: $\frac{m-n}{m}$. Weitere Kosten: 0.

2. *Fall*: $\texttt{T}[h(y,0)]$ ist **nicht leer**.

Wahrscheinlichkeit: $\frac{n}{m}$. Weitere Kosten: $\boldsymbol{C'_{m-1,n-1}}$.

(Suche in $m-1$ Plätzen, von denen $n-1$ besetzt sind.)

Rekursionsformel, für $0 \leq n < m$:

Beweis von (i): $x_1, \ldots, x_n$ sind gespeichert, $y \notin \{x_1, \ldots, x_n\}$ wird gesucht.

In jedem Fall wird $\texttt{T}[h(y,0)]$ untersucht, Kosten 1.

1. *Fall*: $\texttt{T}[h(y,0)]$ ist leer.

Wahrscheinlichkeit: $\frac{m-n}{m}$. Weitere Kosten: 0.

2. *Fall*: $\texttt{T}[h(y,0)]$ ist **nicht leer**.

Wahrscheinlichkeit: $\frac{n}{m}$. Weitere Kosten: $\boldsymbol{C'_{m-1,n-1}}$.

(Suche in $m-1$ Plätzen, von denen $n-1$ besetzt sind.)

Rekursionsformel, für $0 \leq n < m$:

$$C'_{m,n} = \begin{cases} 1 & \text{, falls } n = 0 \end{cases}$$

Beweis von (i): $x_1, \ldots, x_n$ sind gespeichert, $y \notin \{x_1, \ldots, x_n\}$ wird gesucht.

In jedem Fall wird $\texttt{T}[h(y,0)]$ untersucht, Kosten 1.

1. *Fall*: $\texttt{T}[h(y,0)]$ ist leer.

Wahrscheinlichkeit: $\frac{m-n}{m}$. Weitere Kosten: 0.

2. *Fall*: $\texttt{T}[h(y,0)]$ ist **nicht leer**.

Wahrscheinlichkeit: $\frac{n}{m}$. Weitere Kosten: $\boldsymbol{C'_{m-1,n-1}}$.

(Suche in $m-1$ Plätzen, von denen $n-1$ besetzt sind.)

Rekursionsformel, für $0 \leq n < m$:

$$C'_{m,n} = \begin{cases} 1 & \text{, falls } n = 0 \\ 1 + \frac{n}{m} \cdot C'_{m-1,n-1} & \text{, falls } n \geq 1. \end{cases}$$

Wir haben: $C'_{m,n} = \begin{cases} 1 & \text{, falls } n = 0 \end{cases}$

Wir haben: $$C'_{m,n} = \begin{cases} 1 & \text{, falls } n = 0 \\ 1 + \frac{n}{m} \cdot C'_{m-1,n-1} & \text{, falls } n \geq 1. \end{cases}$$

Wir haben: $C'_{m,n} = \begin{cases} 1 & \text{, falls } n = 0 \\ 1 + \frac{n}{m} \cdot C'_{m-1,n-1} & \text{, falls } n \geq 1. \end{cases}$ Also:

$$C'_{m,0} \quad =$$

Wir haben: $C'_{m,n} = \begin{cases} 1 & \text{, falls } n = 0 \\ 1 + \frac{n}{m} \cdot C'_{m-1,n-1} & \text{, falls } n \geq 1. \end{cases}$ Also:

$$C'_{m,0} = 1;$$

Wir haben: $C'_{m,n} = \begin{cases} 1 & \text{, falls } n = 0 \\ 1 + \frac{n}{m} \cdot C'_{m-1,n-1} & \text{, falls } n \geq 1. \end{cases}$ Also:

$$C'_{m,0} = 1;$$

$$C'_{m,1} =$$

Wir haben: $C'_{m,n} = \begin{cases} 1 & \text{, falls } n = 0 \\ 1 + \frac{n}{m} \cdot C'_{m-1,n-1} & \text{, falls } n \geq 1. \end{cases}$ Also:

$$\begin{aligned} C'_{m,0} &= 1\,; \\ C'_{m,1} &= 1 + \frac{1}{m} \cdot 1 \end{aligned}$$

Wir haben: $C'_{m,n} = \begin{cases} 1 & \text{, falls } n = 0 \\ 1 + \frac{n}{m} \cdot C'_{m-1,n-1} & \text{, falls } n \geq 1. \end{cases}$ Also:

$$\begin{aligned} C'_{m,0} &= 1\,; \\ C'_{m,1} &= 1 + \frac{1}{m} \cdot 1 = \frac{m+1}{m}\,; \end{aligned}$$

Wir haben: $C'_{m,n} = \begin{cases} 1 & \text{, falls } n = 0 \\ 1 + \frac{n}{m} \cdot C'_{m-1,n-1} & \text{, falls } n \geq 1. \end{cases}$ Also:

$$C'_{m,0} = 1;$$

$$C'_{m,1} = 1 + \frac{1}{m} \cdot 1 = \frac{m+1}{m};$$

$$C'_{m,2} =$$

Wir haben: $C'_{m,n} = \begin{cases} 1 & \text{, falls } n = 0 \\ 1 + \frac{n}{m} \cdot C'_{m-1,n-1} & \text{, falls } n \geq 1. \end{cases}$ Also:

$$
\begin{aligned}
C'_{m,0} &= 1\,; \\
C'_{m,1} &= 1 + \frac{1}{m} \cdot 1 = \frac{m+1}{m}\,; \\
C'_{m,2} &= 1 + \frac{2}{m} \cdot \frac{m}{m-1}
\end{aligned}
$$

Wir haben: $C'_{m,n} = \begin{cases} 1 & \text{, falls } n = 0 \\ 1 + \frac{n}{m} \cdot C'_{m-1,n-1} & \text{, falls } n \geq 1. \end{cases}$ Also:

$$
\begin{aligned}
C'_{m,0} &= 1\,; \\
C'_{m,1} &= 1 + \frac{1}{m} \cdot 1 = \frac{m+1}{m}\,; \\
C'_{m,2} &= 1 + \frac{2}{m} \cdot \frac{m}{m-1} = \frac{m+1}{m-1}\,;
\end{aligned}
$$

Wir haben: $C'_{m,n} = \begin{cases} 1 & \text{, falls } n = 0 \\ 1 + \frac{n}{m} \cdot C'_{m-1,n-1} & \text{, falls } n \geq 1. \end{cases}$ Also:

$$
\begin{aligned}
C'_{m,0} &= 1\,; \\
C'_{m,1} &= 1 + \frac{1}{m} \cdot 1 = \frac{m+1}{m}\,; \\
C'_{m,2} &= 1 + \frac{2}{m} \cdot \frac{m}{m-1} = \frac{m+1}{m-1}\,; \\
C'_{m,3} &=
\end{aligned}
$$

Wir haben: $C'_{m,n} = \begin{cases} 1 & \text{, falls } n = 0 \\ 1 + \frac{n}{m} \cdot C'_{m-1,n-1} & \text{, falls } n \geq 1. \end{cases}$ Also:

$$
\begin{aligned}
C'_{m,0} &= 1\,; \\
C'_{m,1} &= 1 + \frac{1}{m} \cdot 1 = \frac{m+1}{m}\,; \\
C'_{m,2} &= 1 + \frac{2}{m} \cdot \frac{m}{m-1} = \frac{m+1}{m-1}\,; \\
C'_{m,3} &= 1 + \frac{3}{m} \cdot \frac{m}{m-2}
\end{aligned}
$$

Wir haben: $C'_{m,n} = \begin{cases} 1 & \text{, falls } n = 0 \\ 1 + \frac{n}{m} \cdot C'_{m-1,n-1} & \text{, falls } n \geq 1. \end{cases}$ Also:

$$
\begin{aligned}
C'_{m,0} &= 1\,; \\
C'_{m,1} &= 1 + \frac{1}{m} \cdot 1 = \frac{m+1}{m}\,; \\
C'_{m,2} &= 1 + \frac{2}{m} \cdot \frac{m}{m-1} = \frac{m+1}{m-1}\,; \\
C'_{m,3} &= 1 + \frac{3}{m} \cdot \frac{m}{m-2} = \frac{m+1}{m-2}\,;
\end{aligned}
$$

Wir haben: $C'_{m,n} = \begin{cases} 1 & \text{, falls } n = 0 \\ 1 + \frac{n}{m} \cdot C'_{m-1,n-1} & \text{, falls } n \geq 1. \end{cases}$ Also:

$$
\begin{aligned}
C'_{m,0} &= 1\,; \\
C'_{m,1} &= 1 + \frac{1}{m} \cdot 1 = \frac{m+1}{m}\,; \\
C'_{m,2} &= 1 + \frac{2}{m} \cdot \frac{m}{m-1} = \frac{m+1}{m-1}\,; \\
C'_{m,3} &= 1 + \frac{3}{m} \cdot \frac{m}{m-2} = \frac{m+1}{m-2}\,;
\end{aligned}
$$

und so weiter.

Wir haben: $C'_{m,n} = \begin{cases} 1 & \text{, falls } n = 0 \\ 1 + \frac{n}{m} \cdot C'_{m-1,n-1} & \text{, falls } n \geq 1. \end{cases}$ Also:

$$\begin{aligned} C'_{m,0} &= 1\,; \\ C'_{m,1} &= 1 + \frac{1}{m} \cdot 1 = \frac{m+1}{m}\,; \\ C'_{m,2} &= 1 + \frac{2}{m} \cdot \frac{m}{m-1} = \frac{m+1}{m-1}\,; \\ C'_{m,3} &= 1 + \frac{3}{m} \cdot \frac{m}{m-2} = \frac{m+1}{m-2}\,; \end{aligned}$$

und so weiter. Dies liefert:

$$C'_{m,n} = \frac{m+1}{m+1-n}.$$

Wir haben: $C'_{m,n} = \begin{cases} 1 & \text{, falls } n = 0 \\ 1 + \frac{n}{m} \cdot C'_{m-1,n-1} & \text{, falls } n \geq 1. \end{cases}$ Also:

$$\begin{aligned} C'_{m,0} &= 1\,; \\ C'_{m,1} &= 1 + \frac{1}{m} \cdot 1 = \frac{m+1}{m}\,; \\ C'_{m,2} &= 1 + \frac{2}{m} \cdot \frac{m}{m-1} = \frac{m+1}{m-1}\,; \\ C'_{m,3} &= 1 + \frac{3}{m} \cdot \frac{m}{m-2} = \frac{m+1}{m-2}\,; \end{aligned}$$

und so weiter. Dies liefert:

$$C'_{m,n} = \frac{m+1}{m+1-n}.$$

(Formaler Beweis durch vollständige Induktion.) □

Satz 5.4.5

Unter der Annahme (IH) gilt bei $\alpha = n/m$:

(i) $C'_{m,n} = \frac{m+1}{m+1-n} = \frac{1}{1-\frac{n}{m+1}}$.

Dies ist $< \frac{1}{1-n/m} = \frac{1}{1-\alpha}$.

Wenn α fest und $n, m \to \infty$, dann gilt $C'_{m,n} \to \frac{1}{1-\alpha}$.

Satz 5.4.5

Unter der Annahme (IH) gilt bei $\alpha = n/m$:

(i) $C'_{m,n} = \frac{m+1}{m+1-n} = \frac{1}{1-\frac{n}{m+1}}$.

Dies ist $< \frac{1}{1-n/m} = \frac{1}{1-\alpha}$.

Wenn α fest und $n, m \to \infty$, dann gilt $C'_{m,n} \to \frac{1}{1-\alpha}$.

(ii) Für $\boldsymbol{T_{m,n}}$ = #(untersuchte Fächer bei **erfolgreicher Suche**), gemittelt über $x_1, \ldots, x_n$, gilt:

Satz 5.4.5

Unter der Annahme (IH) gilt bei $\alpha = n/m$:

(i) $C'_{m,n} = \frac{m+1}{m+1-n} = \frac{1}{1-\frac{n}{m+1}}$.

Dies ist $< \frac{1}{1-n/m} = \frac{1}{1-\alpha}$.

Wenn α fest und $n, m \to \infty$, dann gilt $C'_{m,n} \to \frac{1}{1-\alpha}$.

(ii) Für $\boldsymbol{T_{m,n}}$ = #(untersuchte Fächer bei **erfolgreicher Suche**), gemittelt über $x_1, \ldots, x_n$, gilt:

$\boldsymbol{C_{m,n}}$

Satz 5.4.5

Unter der Annahme (IH) gilt bei $\alpha = n/m$:

(i) $C'_{m,n} = \frac{m+1}{m+1-n} = \frac{1}{1-\frac{n}{m+1}}$.

Dies ist $< \frac{1}{1-n/m} = \frac{1}{1-\alpha}$.

Wenn α fest und $n, m \to \infty$, dann gilt $C'_{m,n} \to \frac{1}{1-\alpha}$.

(ii) Für $\boldsymbol{T_{m,n}} = \#$(untersuchte Fächer bei **erfolgreicher Suche**), gemittelt über $x_1, \ldots, x_n$, gilt:

$\boldsymbol{C_{m,n}} = \mathbf{E}(T_{m,n})$

Satz 5.4.5

Unter der Annahme (IH) gilt bei $\alpha = n/m$:

(i) $C'_{m,n} = \frac{m+1}{m+1-n} = \frac{1}{1-\frac{n}{m+1}}$.

Dies ist $< \frac{1}{1-n/m} = \frac{1}{1-\alpha}$.

Wenn α fest und $n, m \to \infty$, dann gilt $C'_{m,n} \to \frac{1}{1-\alpha}$.

(ii) Für $\boldsymbol{T_{m,n}} = \#$(untersuchte Fächer bei **erfolgreicher Suche**), gemittelt über $x_1, \ldots, x_n$, gilt:

$\boldsymbol{C_{m,n}} = \mathbf{E}(T_{m,n}) < \frac{1}{\alpha} \cdot \ln(\frac{1}{1-\alpha})$.

Beweis von (ii): Beachte: Suchaufwand für $x_i \in S \;=\;$ Einfügeaufwand für x_i.

Beweis von (ii): Beachte: Suchaufwand für $x_i \in S$ $=$ Einfügeaufwand für x_i.

Also Mittelung über $C'_{m,0}, C'_{m,1}, \ldots, C'_{m,n-1}$:

Beweis von (ii): Beachte: Suchaufwand für $x_i \in S$ $=$ Einfügeaufwand für x_i.

Also Mittelung über $C'_{m,0}, C'_{m,1}, \ldots, C'_{m,n-1}$:

$$C_{m,n} = \frac{1}{n}$$

Beweis von (ii): Beachte: Suchaufwand für $x_i \in S$ = Einfügeaufwand für x_i.

Also Mittelung über $C'_{m,0}, C'_{m,1}, \ldots, C'_{m,n-1}$:

$$C_{m,n} = \frac{1}{n} \cdot \sum_{1 \leq i \leq n}$$

Beweis von (ii): Beachte: Suchaufwand für $x_i \in S$ = Einfügeaufwand für x_i.

Also Mittelung über $C'_{m,0}, C'_{m,1}, \ldots, C'_{m,n-1}$:

$$C_{m,n} = \frac{1}{n} \cdot \sum_{1 \leq i \leq n} \frac{m+1}{m+1-(i-1)}$$

Beweis von (ii): Beachte: Suchaufwand für $x_i \in S$ = Einfügeaufwand für x_i.

Also Mittelung über $C'_{m,0}, C'_{m,1}, \ldots, C'_{m,n-1}$:

$$C_{m,n} = \frac{1}{n} \cdot \sum_{1 \leq i \leq n} \frac{m+1}{m+1-(i-1)}$$

$$= \frac{m+1}{n}$$

Beweis von (ii): Beachte: Suchaufwand für $x_i \in S$ = Einfügeaufwand für x_i.

Also Mittelung über $C'_{m,0}, C'_{m,1}, \ldots, C'_{m,n-1}$:

$$C_{m,n} = \frac{1}{n} \cdot \sum_{1 \le i \le n} \frac{m+1}{m+1-(i-1)}$$

$$= \frac{m+1}{n} \cdot \left(\frac{1}{m+1} + \frac{1}{m} + \cdots + \frac{1}{m+1-(n-1)} \right).$$

Beweis von (ii): Beachte: Suchaufwand für $x_i \in S$ = Einfügeaufwand für x_i.

Also Mittelung über $C'_{m,0}, C'_{m,1}, \ldots, C'_{m,n-1}$:

$$C_{m,n} = \frac{1}{n} \cdot \sum_{1 \le i \le n} \frac{m+1}{m+1-(i-1)}$$

$$= \frac{m+1}{n} \cdot \left(\frac{1}{m+1} + \frac{1}{m} + \cdots + \frac{1}{m+1-(n-1)} \right).$$

Leicht zu sehen (Untersumme!):

Beweis von (ii): Beachte: Suchaufwand für $x_i \in S$ = Einfügeaufwand für x_i.

Also Mittelung über $C'_{m,0}, C'_{m,1}, \ldots, C'_{m,n-1}$:

$$C_{m,n} = \frac{1}{n} \cdot \sum_{1 \le i \le n} \frac{m+1}{m+1-(i-1)}$$

$$= \frac{m+1}{n} \cdot \left(\frac{1}{m+1} + \frac{1}{m} + \cdots + \frac{1}{m+1-(n-1)} \right).$$

Leicht zu sehen (Untersumme!):

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

Beweis von (ii): Beachte: Suchaufwand für $x_i \in S$ = Einfügeaufwand für x_i.
Also Mittelung über $C'_{m,0}, C'_{m,1}, \ldots, C'_{m,n-1}$:

$$C_{m,n} = \frac{1}{n} \cdot \sum_{1 \le i \le n} \frac{m+1}{m+1-(i-1)}$$

$$= \frac{m+1}{n} \cdot \left(\frac{1}{m+1} + \frac{1}{m} + \cdots + \frac{1}{m+1-(n-1)} \right).$$

Leicht zu sehen (Untersumme!):

$$\frac{1}{m+1} + \frac{1}{m} + \cdots + \frac{1}{m+1-(n-1)} \le \int_{m+1-n}^{m+1} \frac{dt}{t} =$$

Beweis von (ii): Beachte: Suchaufwand für $x_i \in S$ = Einfügeaufwand für x_i.

Also Mittelung über $C'_{m,0}, C'_{m,1}, \ldots, C'_{m,n-1}$:

$$C_{m,n} = \frac{1}{n} \cdot \sum_{1 \le i \le n} \frac{m+1}{m+1-(i-1)}$$

$$= \frac{m+1}{n} \cdot \left(\frac{1}{m+1} + \frac{1}{m} + \cdots + \frac{1}{m+1-(n-1)} \right).$$

Leicht zu sehen (Untersumme!):

$$\frac{1}{m+1} + \frac{1}{m} + \cdots + \frac{1}{m+1-(n-1)} \le \int_{m+1-n}^{m+1} \frac{dt}{t} =$$

$$= \ln(m+1) - \ln(m+1-n) = \ln\left(\frac{m+1}{m+1-n}\right).$$

Also:

$$C_{m,n} \leq \frac{m+1}{n} \cdot \ln\left(\frac{1}{1-\frac{n}{m+1}}\right) \overset{(!)}{<} \frac{m}{n} \cdot \ln\left(\frac{1}{1-\frac{n}{m}}\right).$$

Also:

$$C_{m,n} \leq \frac{m+1}{n} \cdot \ln\left(\frac{1}{1-\frac{n}{m+1}}\right) \overset{(!)}{<} \frac{m}{n} \cdot \ln\left(\frac{1}{1-\frac{n}{m}}\right).$$

Für $\alpha = n/m$, α fest, $n, m \to \infty$, ist $C_{m,n} \approx \frac{1}{\alpha} \ln(\frac{1}{1-\alpha})$.

Also:

$$C_{m,n} \leq \frac{m+1}{n} \cdot \ln\left(\frac{1}{1-\frac{n}{m+1}}\right) \overset{(!)}{<} \frac{m}{n} \cdot \ln\left(\frac{1}{1-\frac{n}{m}}\right).$$

Für $\alpha = n/m$, α fest, $n, m \to \infty$, ist $C_{m,n} \approx \frac{1}{\alpha} \ln(\frac{1}{1-\alpha})$. □

Also:

$$C_{m,n} \leq \frac{m+1}{n} \cdot \ln\left(\frac{1}{1-\frac{n}{m+1}}\right) \overset{(!)}{<} \frac{m}{n} \cdot \ln\left(\frac{1}{1-\frac{n}{m}}\right).$$

Für $\alpha = n/m$, α fest, $n, m \to \infty$, ist $C_{m,n} \approx \frac{1}{\alpha} \ln(\frac{1}{1-\alpha})$. □

Für „$\overset{(!)}{<}$“ beweist man durch Kurvendiskussion: $t \mapsto \frac{1}{t} \cdot \ln(\frac{1}{1-t})$ ist für $0 < t < 1$ monoton wachsend.

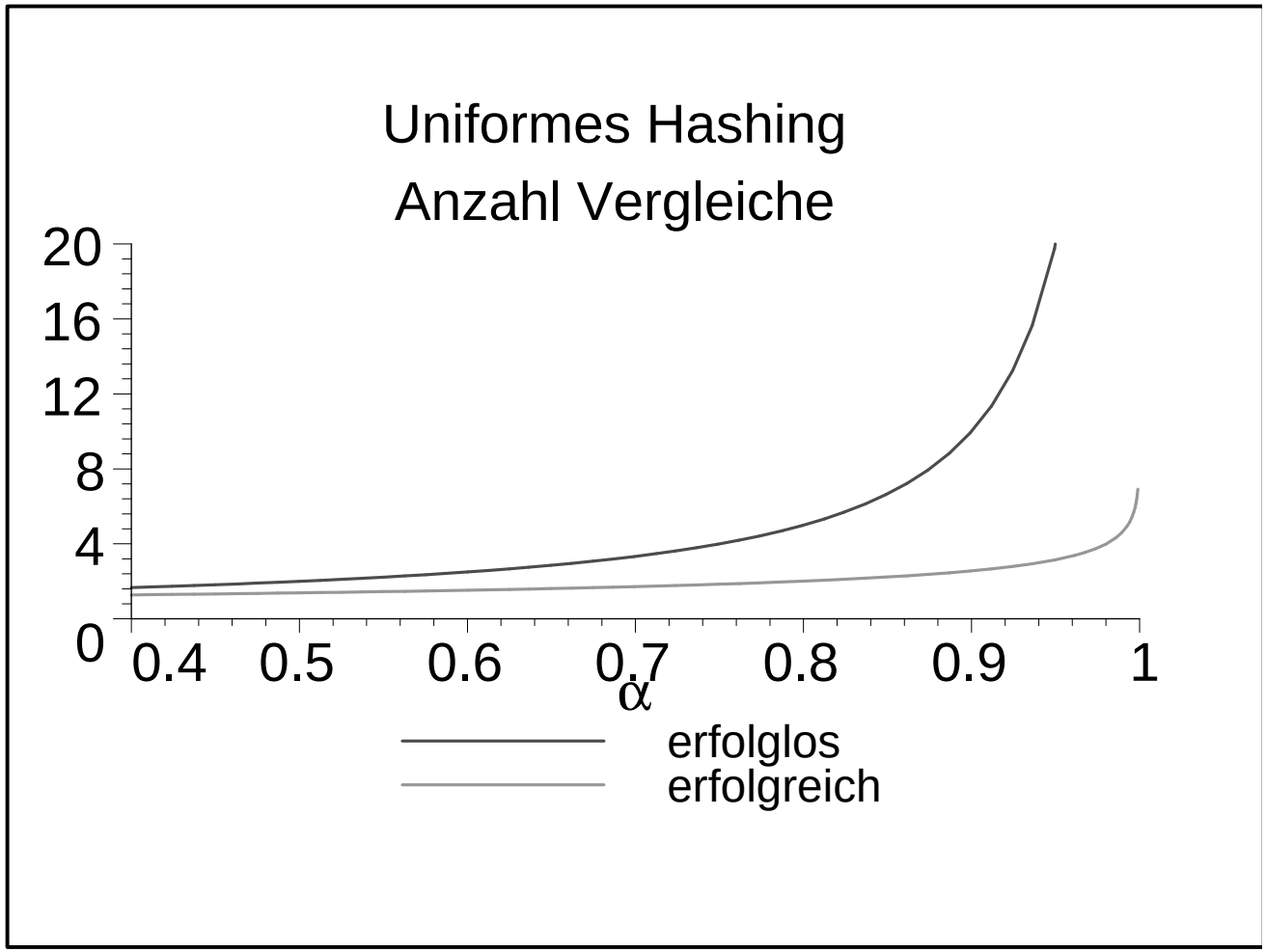
Also:

$$C_{m,n} \leq \frac{m+1}{n} \cdot \ln\left(\frac{1}{1-\frac{n}{m+1}}\right) \overset{(!)}{<} \frac{m}{n} \cdot \ln\left(\frac{1}{1-\frac{n}{m}}\right).$$

Für $\alpha = n/m$, α fest, $n, m \to \infty$, ist $C_{m,n} \approx \frac{1}{\alpha} \ln(\frac{1}{1-\alpha})$. □

Für „$\overset{(!)}{<}$“ beweist man durch Kurvendiskussion: $t \mapsto \frac{1}{t} \cdot \ln(\frac{1}{1-t})$ ist für $0 < t < 1$ monoton wachsend.

Bemerkung: Für $\alpha \nearrow 1$ geht $\frac{1}{\alpha} \cdot \ln(\frac{1}{1-\alpha})$ **extrem langsam** gegen ∞.


Uniformes Hashing
Anzahl Vergleiche
20
16
12
8
4
0
0.4
0.5
0.6
0.7
0.8
0.9
1
α
erfolglos
erfolgreich

Erwartete Anzahl von Schlüsselvergleichen bei uniformem Hashing:

Erwartete Anzahl von Schlüsselvergleichen bei uniformem Hashing:

	erfolglose Suche	erfolgreiche Suche (Mittel über $x_1, \ldots, x_n$)
α	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha} \ln(\frac{1}{1-\alpha})$

Erwartete Anzahl von Schlüsselvergleichen bei uniformem Hashing:

	erfolglose Suche	erfolgreiche Suche (Mittel über $x_1, \ldots, x_n$)
α	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha}\ln(\frac{1}{1-\alpha})$
0,5	2	1,39

Erwartete Anzahl von Schlüsselvergleichen bei uniformem Hashing:

	erfolglose Suche	erfolgreiche Suche (Mittel über $x_1, \ldots, x_n$)
α	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha}\ln(\frac{1}{1-\alpha})$
0,5	2	1,39
0,6	2,5	1,53

Erwartete Anzahl von Schlüsselvergleichen bei uniformem Hashing:

	erfolglose Suche	erfolgreiche Suche (Mittel über $x_1, \ldots, x_n$)
α	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha}\ln(\frac{1}{1-\alpha})$
0,5	2	1,39
0,6	2,5	1,53
0,7	3,33	1,72

Erwartete Anzahl von Schlüsselvergleichen bei uniformem Hashing:

	erfolglose Suche	erfolgreiche Suche (Mittel über $x_1, \ldots, x_n$)
α	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha}\ln(\frac{1}{1-\alpha})$
0,5	2	1,39
0,6	2,5	1,53
0,7	3,33	1,72
0,75	4	1,96

Erwartete Anzahl von Schlüsselvergleichen bei uniformem Hashing:

	erfolglose Suche	erfolgreiche Suche (Mittel über $x_1, \ldots, x_n$)
α	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha}\ln(\frac{1}{1-\alpha})$
0,5	2	1,39
0,6	2,5	1,53
0,7	3,33	1,72
0,75	4	1,96
0,8	5	2,01

Erwartete Anzahl von Schlüsselvergleichen bei uniformem Hashing:

	erfolglose Suche	erfolgreiche Suche (Mittel über $x_1, \ldots, x_n$)
α	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha}\ln(\frac{1}{1-\alpha})$
0,5	2	1,39
0,6	2,5	1,53
0,7	3,33	1,72
0,75	4	1,96
0,8	5	2,01
0,9	10	2,56

Erwartete Anzahl von Schlüsselvergleichen bei uniformem Hashing:

	erfolglose Suche	erfolgreiche Suche (Mittel über $x_1, \ldots, x_n$)
α	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha}\ln(\frac{1}{1-\alpha})$
0,5	2	1,39
0,6	2,5	1,53
0,7	3,33	1,72
0,75	4	1,96
0,8	5	2,01
0,9	10	2,56
0,95	**20**	3,15

Erwartete Anzahl von Schlüsselvergleichen bei uniformem Hashing:

	erfolglose Suche	erfolgreiche Suche (Mittel über $x_1, \ldots, x_n$)
α	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha}\ln(\frac{1}{1-\alpha})$
0,5	2	1,39
0,6	2,5	1,53
0,7	3,33	1,72
0,75	4	1,96
0,8	5	2,01
0,9	10	2,56
0,95	**20**	3,15
0,98	**50**	3,99

Erwartete Anzahl von Schlüsselvergleichen bei uniformem Hashing:

	erfolglose Suche	erfolgreiche Suche (Mittel über $x_1, \ldots, x_n$)
α	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha}\ln(\frac{1}{1-\alpha})$
0,5	2	1,39
0,6	2,5	1,53
0,7	3,33	1,72
0,75	4	1,96
0,8	5	2,01
0,9	10	2,56
0,95	**20**	3,15
0,98	**50**	3,99
0,99	**100**	4,65

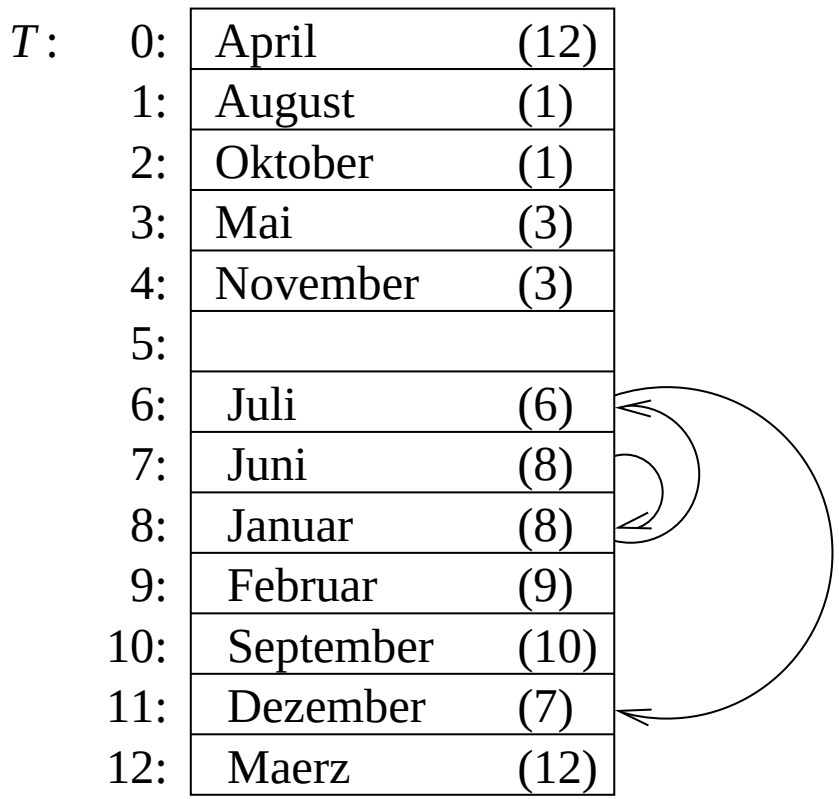
Vorlesungsvideo:

Löschen bei geschlossenem Hashing

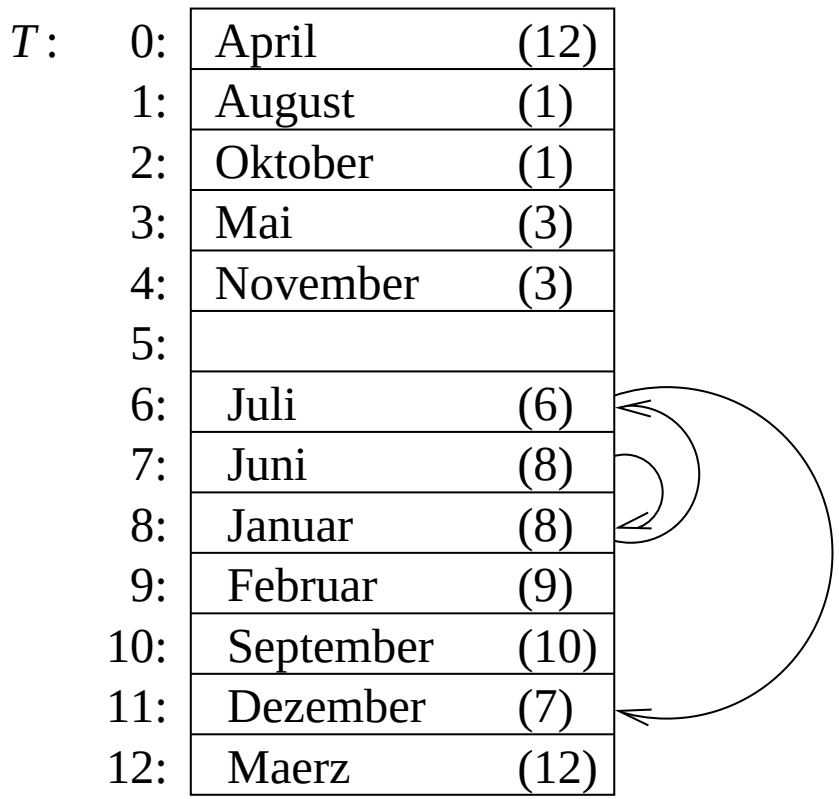
Löschungen?

Löschungen? Problem z. B. bei Quadratischem Sondieren.

Löschungen? Problem z. B. bei Quadratischem Sondieren.

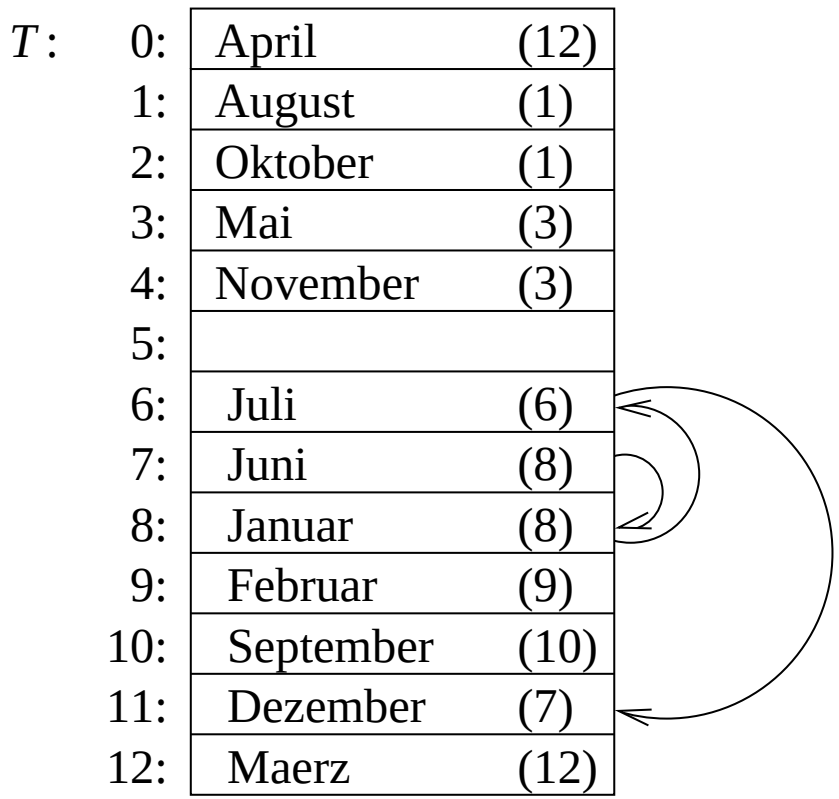


Löschungen? Problem z. B. bei Quadratischem Sondieren.



Suchfolge „Dezember“ mit
$h(\text{Dezember}) = 7$: $7, 8, 6, 11$.

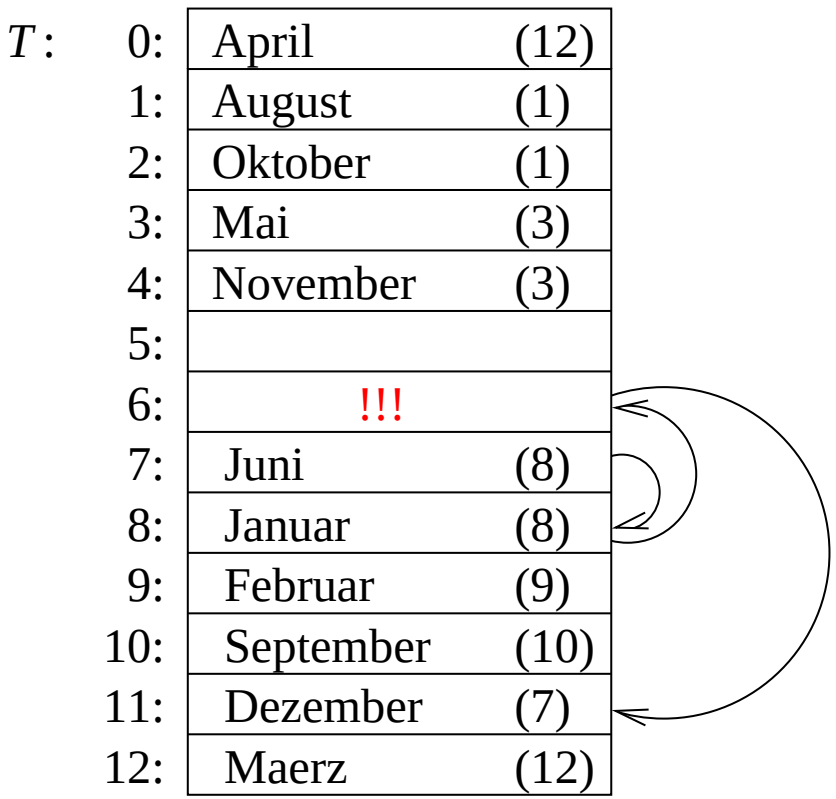
Löschungen? Problem z. B. bei Quadratischem Sondieren.



Suchfolge „Dezember“ mit
$h(\text{Dezember}) = 7$: $7, 8, 6, 11$.

Lösche „Juli“!

Löschungen? Problem z. B. bei Quadratischem Sondieren.



Suchfolge „Dezember“ mit
$h(\text{Dezember}) = 7$: $7, 8, 6, 11$.

Lösche „Juli“!

„Dezember“ wird nicht mehr gefunden.

Ausweg: Jede Zelle hat Statusvariable `status` (*voll*, *leer*, *gelöscht*).

Ausweg: Jede Zelle hat Statusvariable `status` (*voll*, *leer*, *gelöscht*).

Löschen: Zelle nicht auf *leer*, sondern auf *gelöscht* setzen (engl.: „*tombstone*“).

Ausweg: Jede Zelle hat Statusvariable `status` (*voll*, *leer*, *gelöscht*).

Löschen: Zelle nicht auf *leer*, sondern auf *gelöscht* setzen (engl.: „*tombstone*“).

Suchen: Immer bis zur ersten *leer*en Zelle gehen.

Ausweg: Jede Zelle hat Statusvariable `status` (*voll*, *leer*, *gelöscht*).

Löschen: Zelle nicht auf *leer*, sondern auf *gelöscht* setzen (engl.: „*tombstone*“).

Suchen: Immer bis zur ersten *leer*en Zelle gehen.

Zellen, die *gelöscht* sind, dürfen mit einem neuen Schlüssel überschrieben werden.

Ausweg: Jede Zelle hat Statusvariable `status` (*voll*, *leer*, *gelöscht*).

Löschen: Zelle nicht auf *leer*, sondern auf *gelöscht* setzen (engl.: „*tombstone*“).

Suchen: Immer bis zur ersten *leer*en Zelle gehen.

Zellen, die *gelöscht* sind, dürfen mit einem neuen Schlüssel überschrieben werden.

Überlauf tritt **spätestens** ein, wenn die letzte *leer*e Zelle beschrieben werden würde.

Ausweg: Jede Zelle hat Statusvariable `status` (*voll*, *leer*, *gelöscht*).

Löschen: Zelle nicht auf *leer*, sondern auf *gelöscht* setzen (engl.: „*tombstone*“).

Suchen: Immer bis zur ersten *leer*en Zelle gehen.

Zellen, die *gelöscht* sind, dürfen mit einem neuen Schlüssel überschrieben werden.

Überlauf tritt **spätestens** ein, wenn die letzte *leer*e Zelle beschrieben werden würde.

Sinnvoller: Überlauf tritt ein, wenn die Anzahl der *voll*en und *gelöscht*en Zellen zusammen eine feste Schranke `bound` überschreitet.

Ausweg: Jede Zelle hat Statusvariable `status` (*voll*, *leer*, *gelöscht*).

Löschen: Zelle nicht auf *leer*, sondern auf *gelöscht* setzen (engl.: „*tombstone*“).

Suchen: Immer bis zur ersten *leer*en Zelle gehen.

Zellen, die *gelöscht* sind, dürfen mit einem neuen Schlüssel überschrieben werden.

Überlauf tritt **spätestens** ein, wenn die letzte *leer*e Zelle beschrieben werden würde.

Sinnvoller: Überlauf tritt ein, wenn die Anzahl der *voll*en und *gelöscht*en Zellen zusammen eine feste Schranke `bound` überschreitet.

Z.B. $m - 1$ oder $\alpha_1 m$, α_1 passend zum Sondierungsverfahren.

Alle Operationen bei offener Adressierung

empty(m):

Alle Operationen bei offener Adressierung

empty(m):

Lege leeres Array $\texttt{T}[0\,..\,m-1]$

Alle Operationen bei offener Adressierung

empty(m):

Lege leeres Array `T`$[0\,..\,m-1]$ an.

Alle Operationen bei offener Adressierung

empty(m):

Lege leeres Array `T`$[0\,..\,m-1]$ an.

Initialisiere alle Zellen `T`$[j]$ mit $(-,-,$ *leer*$)$.

Alle Operationen bei offener Adressierung

empty(m):

Lege leeres Array `T`$[0\,..\,m-1]$ an.

Initialisiere alle Zellen `T`$[j]$ mit $(-,-,$ *leer*$)$.

Wähle Hashfunktion $h\colon U \times [m] \to [m]$.

Initialisiere `inUse` mit 0.

Alle Operationen bei offener Adressierung

empty(m):

Lege leeres Array `T`$[0\,..\,m-1]$ an.

Initialisiere alle Zellen `T`$[j]$ mit $(-,-,$ *leer*$)$.

Wähle Hashfunktion $h\colon U \times [m] \to [m]$.

Initialisiere `inUse` mit 0. (Zählt **nicht-leere** Zellen.)

Alle Operationen bei offener Adressierung

empty(m):

Lege leeres Array `T`$[0\,..\,m-1]$ an.

Initialisiere alle Zellen `T`$[j]$ mit $(-,-,$ *leer*$)$.

Wähle Hashfunktion $h\colon U \times [m] \to [m]$.

Initialisiere `inUse` mit 0. (Zählt **nicht-leere** Zellen.)

Initialisiere `load` mit 0.

Alle Operationen bei offener Adressierung

empty(m):

Lege leeres Array `T`$[0\,..\,m-1]$ an.

Initialisiere alle Zellen `T`$[j]$ mit $(-,-,$ *leer*$)$.

Wähle Hashfunktion $h\colon U \times [m] \to [m]$.

Initialisiere `inUse` mit 0. (Zählt **nicht-leere** Zellen.)

Initialisiere `load` mit 0. (Zählt Einträge.)

Alle Operationen bei offener Adressierung

empty(m):

Lege leeres Array `T`$[0\,..\,m-1]$ an.

Initialisiere alle Zellen `T`$[j]$ mit $(-,-,$ *leer*$)$.

Wähle Hashfunktion $h\colon U \times [m] \to [m]$.

Initialisiere `inUse` mit 0. (Zählt **nicht-leere** Zellen.)

Initialisiere `load` mit 0. (Zählt Einträge.)

Initialisiere `bound` (z. B. mit $\alpha_0 m$, maximal $m-1$).

lookup(x):

lookup(x):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$ mit

1. Fall: `T[`l`].status` $=$ *leer*: **return** „*not found*“;

lookup(x):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$ mit

1. Fall: `T[`l`].status` $=$ *leer*: **return** „*not found*“;

oder

2. Fall: (`T[`l`].status` $=$ *voll* **und** `T[`l`].key` $= x$):

lookup(x):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$ mit

1. Fall: `T[`l`].status` $=$ *leer*: **return** „*not found*“;

oder

2. Fall: (`T[`l`].status` $=$ *voll* **und** `T[`l`].key` $= x$): **return** `T[`l`].data`.

lookup(x):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$ mit

1. Fall: `T[`l`].status` $=$ *leer*: **return** „*not found*“;

oder

2. Fall: (`T[`l`].status` $=$ *voll* **und** `T[`l`].key` $= x$): **return** `T[`l`].data`.

delete(x):

lookup(x):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$ mit

1. Fall: `T[`l`].status` $=$ *leer*: **return** „*not found*“;

oder

2. Fall: (`T[`l`].status` $=$ *voll* **und** `T[`l`].key` $= x$): **return** `T[`l`].data`.

delete(x):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$ mit

1. Fall: `T[`l`].status` $=$ *leer*:

lookup(x):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$ mit

1. Fall: `T[`l`].status` $=$ *leer*: **return** „*not found*“;

oder

2. Fall: (`T[`l`].status` $=$ *voll* **und** `T[`l`].key` $= x$): **return** `T[`l`].data`.

delete(x):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$ mit

1. Fall: `T[`l`].status` $=$ *leer*: tue nichts; // evtl. Warnung: *not found*

lookup(x):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$ mit

1. Fall: `T[`l`].status` $=$ *leer*: **return** „*not found*“;

oder

2. Fall: (`T[`l`].status` $=$ *voll* **und** `T[`l`].key` $= x$): **return** `T[`l`].data`.

delete(x):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$ mit

1. Fall: `T[`l`].status` $=$ *leer*: tue nichts; // evtl. Warnung: *not found*

oder

2. Fall: (`T[`l`].status` $=$ *voll* **und** `T[`l`].key` $= x$):

lookup(x):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$ mit

1. Fall: `T[`l`].status` $=$ *leer*: **return** „*not found*“;

oder

2. Fall: (`T[`l`].status` $=$ *voll* **und** `T[`l`].key` $= x$): **return** `T[`l`].data`.

delete(x):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$ mit

1. Fall: `T[`l`].status` $=$ *leer*: tue nichts; // evtl. Warnung: *not found*

oder

2. Fall: (`T[`l`].status` $=$ *voll* **und** `T[`l`].key` $= x$): `T[`l`]` $\leftarrow (-, -,$ *gelöscht*$)$; `load--`.

insert(x, r):

insert(x, r):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$
mit `T[`l`].status` $=$ *leer* **oder** (`T[`l`].status` $=$ *voll* **und** `T[`l`].key` $= x$);

insert(x, r):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$
mit `T[`l`].status` $=$ *leer* **oder** (`T[`l`].status` $=$ *voll* **und** `T[`l`].key` $= x$);

Merke dabei das erste l' in dieser Folge mit `T[`l'`].status` $\in \{$*gelöscht*, *leer*$\}$;

insert(x, r):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$
mit $\texttt{T}[l]\texttt{.status} = \textit{leer}$ **oder** $(\texttt{T}[l]\texttt{.status} = \textit{voll}$ **und** $\texttt{T}[l]\texttt{.key} = x)$;

Merke dabei das erste l' in dieser Folge mit $\texttt{T}[l']\texttt{.status} \in \{\textit{gelöscht}, \textit{leer}\}$;

1. Fall: $\texttt{T}[l]\texttt{.status} = \textit{voll}$:

insert(x, r):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$
mit $\texttt{T}[l]\texttt{.status} = \textit{leer}$ **oder** $(\texttt{T}[l]\texttt{.status} = \textit{voll}$ **und** $\texttt{T}[l]\texttt{.key} = x)$;

Merke dabei das erste l' in dieser Folge mit $\texttt{T}[l']\texttt{.status} \in \{\textit{gelöscht}, \textit{leer}\}$;

1. Fall: $\texttt{T}[l]\texttt{.status} = \textit{voll}$: $\texttt{T}[l]\texttt{.data} \leftarrow r$; // Update

insert(x, r):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$
mit $\texttt{T}[l]\texttt{.status} = \textit{leer}$ **oder** $(\texttt{T}[l]\texttt{.status} = \textit{voll}$ **und** $\texttt{T}[l]\texttt{.key} = x)$;

Merke dabei das erste l' in dieser Folge mit $\texttt{T}[l']\texttt{.status} \in \{\textit{gelöscht}, \textit{leer}\}$;

1. Fall: $\texttt{T}[l]\texttt{.status} = \textit{voll}$: $\texttt{T}[l]\texttt{.data} \leftarrow r$; // Update

2. Fall: $\texttt{T}[l']\texttt{.status} = \textit{gelöscht}$:

insert(x, r):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$
mit `T[`l`].status` $=$ *leer* **oder** (`T[`l`].status` $=$ *voll* **und** `T[`l`].key` $= x$);

Merke dabei das erste l' in dieser Folge mit `T[`l'`].status` $\in \{$*gelöscht*, *leer*$\}$;

1. Fall: `T[`l`].status` $=$ *voll*: `T[`l`].data` $\leftarrow r$; // Update

2. Fall: `T[`l'`].status` $=$ *gelöscht*: `T[`l'`]` $\leftarrow (x, r,$ *voll*$)$; `load++`;

insert(x, r):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$
mit `T[`l`].status` $=$ *leer* **oder** (`T[`l`].status` $=$ *voll* **und** `T[`l`].key` $= x$);

Merke dabei das erste l' in dieser Folge mit `T[`l'`].status` $\in \{$*gelöscht*, *leer*$\}$;

1. Fall: `T[`l`].status` $=$ *voll*: `T[`l`].data` $\leftarrow r$; // Update

2. Fall: `T[`l'`].status` $=$ *gelöscht*: `T[`l'`]` $\leftarrow (x, r,$ *voll*$)$; `load++`;
// überschreibe gelöschtes Feld

insert(x, r):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$
mit $\texttt{T}[l]\texttt{.status} = \textit{leer}$ **oder** $(\texttt{T}[l]\texttt{.status} = \textit{voll}$ **und** $\texttt{T}[l]\texttt{.key} = x)$;

Merke dabei das erste l' in dieser Folge mit $\texttt{T}[l']\texttt{.status} \in \{\textit{gelöscht}, \textit{leer}\}$;

1. **Fall:** $\texttt{T}[l]\texttt{.status} = \textit{voll}$: $\texttt{T}[l]\texttt{.data} \leftarrow r$; // Update
2. **Fall:** $\texttt{T}[l']\texttt{.status} = \textit{gelöscht}$: $\texttt{T}[l'] \leftarrow (x, r, \textit{voll})$; `load++`;
   // überschreibe gelöschtes Feld
3. **Fall:** $\texttt{T}[l']\texttt{.status} = \textit{leer}$:

insert(x, r):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$
mit $\texttt{T}[l]\texttt{.status} = \textit{leer}$ **oder** $(\texttt{T}[l]\texttt{.status} = \textit{voll}$ **und** $\texttt{T}[l]\texttt{.key} = x)$;

Merke dabei das erste l' in dieser Folge mit $\texttt{T}[l']\texttt{.status} \in \{\textit{gelöscht}, \textit{leer}\}$;

1. **Fall:** $\texttt{T}[l]\texttt{.status} = \textit{voll}$: $\texttt{T}[l]\texttt{.data} \leftarrow r$; // Update
2. **Fall:** $\texttt{T}[l']\texttt{.status} = \textit{gelöscht}$: $\texttt{T}[l'] \leftarrow (x, r, \textit{voll})$; `load++`;
   // überschreibe gelöschtes Feld
3. **Fall:** $\texttt{T}[l']\texttt{.status} = \textit{leer}$: // $l = l'$, neue Zelle

insert(x, r):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$
mit $\texttt{T}[l]\texttt{.status} = \textit{leer}$ **oder** $(\texttt{T}[l]\texttt{.status} = \textit{voll}$ **und** $\texttt{T}[l]\texttt{.key} = x)$;

Merke dabei das erste l' in dieser Folge mit $\texttt{T}[l']\texttt{.status} \in \{\textit{gelöscht}, \textit{leer}\}$;

1. **Fall:** $\texttt{T}[l]\texttt{.status} = \textit{voll}$: $\texttt{T}[l]\texttt{.data} \leftarrow r$; // Update
2. **Fall:** $\texttt{T}[l']\texttt{.status} = \textit{gelöscht}$: $\texttt{T}[l'] \leftarrow (x, r, \textit{voll})$; `load++`;
   // überschreibe gelöschtes Feld
3. **Fall:** $\texttt{T}[l']\texttt{.status} = \textit{leer}$: // $l = l'$, neue Zelle
   if $\texttt{inUse} + 1 > \texttt{bound}$ **then** „Überlaufbehandlung“

insert(x, r):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$
mit `T[`l`].status` $=$ *leer* **oder** (`T[`l`].status` $=$ *voll* **und** `T[`l`].key` $= x$);

Merke dabei das erste l' in dieser Folge mit `T[`l'`].status` $\in \{$*gelöscht*, *leer*$\}$;

1. **Fall:** `T[`l`].status` $=$ *voll*: `T[`l`].data` $\leftarrow r$; // Update
2. **Fall:** `T[`l'`].status` $=$ *gelöscht*: `T[`l'`]` $\leftarrow (x, r,$ *voll* $)$; `load++`;
   // überschreibe gelöschtes Feld
3. **Fall:** `T[`l'`].status` $=$ *leer*: // $l = l'$, neue Zelle
   if `inUse` $+ 1 >$ `bound` **then** „Überlaufbehandlung“
   else `inUse++`; `load++`; `T[`l`]` $\leftarrow (x, r,$ *voll* $)$;

Überlaufbehandlung, Verdopplung:

Überlaufbehandlung, Verdopplung:

Sinnvoll: **Überlauf** tritt ein, wenn die Anzahl der *voll*en und *gelöscht*en Zellen zusammen (in `inUse`) eine Schranke $\alpha_0 m$ (in `bound`) überschreitet.

Überlaufbehandlung, Verdopplung:

Sinnvoll: **Überlauf** tritt ein, wenn die Anzahl der *voll*en und *gelöscht*en Zellen zusammen (in `inUse`) eine Schranke $\alpha_0 m$ (in `bound`) überschreitet.

(α_0 passend zum Sondierungsverfahren festlegen.)

Überlaufbehandlung, Verdopplung:

Sinnvoll: **Überlauf** tritt ein, wenn die Anzahl der *voll*en und *gelöscht*en Zellen zusammen (in `inUse`) eine Schranke $\alpha_0 m$ (in `bound`) überschreitet.

(α_0 passend zum Sondierungsverfahren festlegen.)

Anzahl der *voll*en Zellen: `load`.

Überlaufbehandlung, Verdopplung:

Sinnvoll: **Überlauf** tritt ein, wenn die Anzahl der *voll*en und *gelöscht*en Zellen zusammen (in `inUse`) eine Schranke $\alpha_0 m$ (in `bound`) überschreitet.

(α_0 passend zum Sondierungsverfahren festlegen.)

Anzahl der *voll*en Zellen: `load`.

Falls $\texttt{load} + 1 > \alpha_1 m$ für (z. B.) $\alpha_1 = 0.8\alpha_0$, **verdopple** Tabellengröße,

Überlaufbehandlung, Verdopplung:

Sinnvoll: **Überlauf** tritt ein, wenn die Anzahl der *voll*en und *gelöscht*en Zellen zusammen (in `inUse`) eine Schranke $\alpha_0 m$ (in `bound`) überschreitet.

(α_0 passend zum Sondierungsverfahren festlegen.)

Anzahl der *voll*en Zellen: `load`.

Falls $\texttt{load} + 1 > \alpha_1 m$ für (z. B.) $\alpha_1 = 0.8\alpha_0$, **verdopple** Tabellengröße, und trage Schlüssel aus `T` **neu** in die größere Tabelle ein. Setze $\texttt{inUse} \leftarrow \texttt{load}$.

Überlaufbehandlung, Verdopplung:

Sinnvoll: **Überlauf** tritt ein, wenn die Anzahl der *voll*en und *gelöscht*en Zellen zusammen (in `inUse`) eine Schranke $\alpha_0 m$ (in `bound`) überschreitet.

(α_0 passend zum Sondierungsverfahren festlegen.)

Anzahl der *voll*en Zellen: `load`.

Falls $\texttt{load} + 1 > \alpha_1 m$ für (z. B.) $\alpha_1 = 0.8\alpha_0$, **verdopple** Tabellengröße, und trage Schlüssel aus `T` **neu** in die größere Tabelle ein. Setze $\texttt{inUse} \leftarrow \texttt{load}$.

Sonst: Trage alle Schlüssel **neu** in die **alte Tabelle** `T` ein,

Überlaufbehandlung, Verdopplung:

Sinnvoll: **Überlauf** tritt ein, wenn die Anzahl der *voll*en und *gelöscht*en Zellen zusammen (in `inUse`) eine Schranke $\alpha_0 m$ (in `bound`) überschreitet.

(α_0 passend zum Sondierungsverfahren festlegen.)

Anzahl der *voll*en Zellen: `load`.

Falls $\texttt{load} + 1 > \alpha_1 m$ für (z. B.) $\alpha_1 = 0.8\alpha_0$, **verdopple** Tabellengröße, und trage Schlüssel aus `T` **neu** in die größere Tabelle ein. Setze $\texttt{inUse} \leftarrow \texttt{load}$.

Sonst: Trage alle Schlüssel **neu** in die **alte Tabelle** `T` ein,
eliminiere alle „*gelöscht*“-Markierungen, wie folgt . . .

Trick für Umorganisieren:

Trick für Umorganisieren: (Ohne zusätzlichen Platz.)

Trick für Umorganisieren: (Ohne zusätzlichen Platz.)

1) `inUse` $\leftarrow$ `load`

Trick für Umorganisieren: (Ohne zusätzlichen Platz.)

1) `inUse` $\leftarrow$ `load`

2) Für $i = 0, 1, \ldots, m-1$ tue folgendes:

Trick für Umorganisieren: (Ohne zusätzlichen Platz.)

1) `inUse` $\leftarrow$ `load`

2) Für $i = 0, 1, \ldots, m-1$ tue folgendes:
Wenn `T[`i`].status` $=$ *voll*, setze `T[`i`].status` $\leftarrow$ *alt*.

Trick für Umorganisieren: (Ohne zusätzlichen Platz.)

1) `inUse` $\leftarrow$ `load`

2) Für $i = 0, 1, \ldots, m-1$ tue folgendes:
Wenn `T[`i`].status` $=$ *voll*, setze `T[`i`].status` $\leftarrow$ *alt*.
Wenn `T[`i`].status` $=$ *gelöscht*, setze `T[`i`].status` $\leftarrow$ *leer*.

Trick für Umorganisieren: (Ohne zusätzlichen Platz.)

1) `inUse` $\leftarrow$ `load`

2) Für $i = 0, 1, \ldots, m-1$ tue folgendes:
Wenn `T[`i`].status` $=$ *voll*, setze `T[`i`].status` $\leftarrow$ *alt*.
Wenn `T[`i`].status` $=$ *gelöscht*, setze `T[`i`].status` $\leftarrow$ *leer*.

3) Für $i = 0, 1, \ldots, m-1$ tue folgendes: // checke jeden Platz

Trick für Umorganisieren: (Ohne zusätzlichen Platz.)

1) `inUse` $\leftarrow$ `load`

2) Für $i = 0, 1, \ldots, m-1$ tue folgendes:
Wenn `T[`i`].status` $=$ *voll*, setze `T[`i`].status` $\leftarrow$ *alt*.
Wenn `T[`i`].status` $=$ *gelöscht*, setze `T[`i`].status` $\leftarrow$ *leer*.

3) Für $i = 0, 1, \ldots, m-1$ tue folgendes: // checke jeden Platz
Wenn `T[`i`].status` $=$ *alt*, dann
    setze $(\texttt{x}, \texttt{r}) \leftarrow (\texttt{T[}i\texttt{].key}, \texttt{T[}i\texttt{].data})$;

Trick für Umorganisieren: (Ohne zusätzlichen Platz.)

1) `inUse` $\leftarrow$ `load`

2) Für $i = 0, 1, \ldots, m-1$ tue folgendes:
Wenn `T[`i`].status` $=$ *voll*, setze `T[`i`].status` $\leftarrow$ *alt*.
Wenn `T[`i`].status` $=$ *gelöscht*, setze `T[`i`].status` $\leftarrow$ *leer*.

3) Für $i = 0, 1, \ldots, m-1$ tue folgendes: // checke jeden Platz
Wenn `T[`i`].status` $=$ *alt*, dann
    setze $(\texttt{x}, \texttt{r}) \leftarrow (\texttt{T[}i\texttt{].key}, \texttt{T[}i\texttt{].data})$;
    setze `T[`i`].status` $\leftarrow$ *leer*;

Trick für Umorganisieren: (Ohne zusätzlichen Platz.)

1) `inUse` $\leftarrow$ `load`

2) Für $i = 0, 1, \ldots, m-1$ tue folgendes:
Wenn `T[`i`].status` $=$ *voll*, setze `T[`i`].status` $\leftarrow$ *alt*.
Wenn `T[`i`].status` $=$ *gelöscht*, setze `T[`i`].status` $\leftarrow$ *leer*.

3) Für $i = 0, 1, \ldots, m-1$ tue folgendes: // checke jeden Platz
Wenn `T[`i`].status` $=$ *alt*, dann
    setze $(\mathtt{x}, \mathtt{r}) \leftarrow (\mathtt{T}[i].\mathtt{key}, \mathtt{T}[i].\mathtt{data})$;
    setze `T[`i`].status` $\leftarrow$ *leer*;
    insert$^*(\mathtt{x}, \mathtt{r})$.

Trick für Umorganisieren: (Ohne zusätzlichen Platz.)

1) `inUse` $\leftarrow$ `load`

2) Für $i = 0, 1, \ldots, m-1$ tue folgendes:
Wenn `T[`i`].status` $=$ *voll*, setze `T[`i`].status` $\leftarrow$ *alt*.
Wenn `T[`i`].status` $=$ *gelöscht*, setze `T[`i`].status` $\leftarrow$ *leer*.

3) Für $i = 0, 1, \ldots, m-1$ tue folgendes: // checke jeden Platz
Wenn `T[`i`].status` $=$ *alt*, dann
    setze $(\mathtt{x}, \mathtt{r}) \leftarrow (\mathtt{T}[i].\mathtt{key}, \mathtt{T}[i].\mathtt{data})$;
    setze `T[`i`].status` $\leftarrow$ *leer*;
    insert$^*(\mathtt{x}, \mathtt{r})$.

    Dabei ist **insert***(x,r) folgende rekursive Variante von **insert**(x,r):
    Finde erstes l in der Folge $h(x, j), j = 0, 1, 2, \ldots$ mit `T[`l`].status` $\in \{$*leer*, *alt*$\}$;

Trick für Umorganisieren: (Ohne zusätzlichen Platz.)

1) `inUse` $\leftarrow$ `load`

2) Für $i = 0, 1, \ldots, m-1$ tue folgendes:
Wenn `T[`i`].status` $=$ *voll*, setze `T[`i`].status` $\leftarrow$ *alt*.
Wenn `T[`i`].status` $=$ *gelöscht*, setze `T[`i`].status` $\leftarrow$ *leer*.

3) Für $i = 0, 1, \ldots, m-1$ tue folgendes: // checke jeden Platz
Wenn `T[`i`].status` $=$ *alt*, dann
    setze $(\mathtt{x}, \mathtt{r}) \leftarrow (\mathtt{T}[i].\mathtt{key}, \mathtt{T}[i].\mathtt{data})$;
    setze `T[`i`].status` $\leftarrow$ *leer*;
    insert$^*(\mathtt{x}, \mathtt{r})$.

    Dabei ist **insert***(x,r) folgende rekursive Variante von **insert**(x,r):
    Finde erstes l in der Folge $h(x, j), j = 0, 1, 2, \ldots$ mit `T[`l`].status` $\in \{$*leer*, *alt*$\}$;
    Falls `T[`l`].status` $=$ *leer*: `T[`l`]` $\leftarrow (x, r,$ *voll*$)$.

Trick für Umorganisieren: (Ohne zusätzlichen Platz.)

1) `inUse` $\leftarrow$ `load`

2) Für $i = 0, 1, \ldots, m-1$ tue folgendes:
Wenn $\texttt{T}[i]\texttt{.status} =$ *voll*, setze $\texttt{T}[i]\texttt{.status} \leftarrow$ *alt*.
Wenn $\texttt{T}[i]\texttt{.status} =$ *gelöscht*, setze $\texttt{T}[i]\texttt{.status} \leftarrow$ *leer*.

3) Für $i = 0, 1, \ldots, m-1$ tue folgendes: // checke jeden Platz
Wenn $\texttt{T}[i]\texttt{.status} =$ *alt*, dann
    setze $(\texttt{x}, \texttt{r}) \leftarrow (\texttt{T}[i]\texttt{.key}, \texttt{T}[i]\texttt{.data})$;
    setze $\texttt{T}[i]\texttt{.status} \leftarrow$ *leer*;
    insert$^*(\texttt{x}, \texttt{r})$.

    Dabei ist **insert***(x,r) folgende rekursive Variante von **insert**(x,r):
    Finde erstes l in der Folge $h(x, j), j = 0, 1, 2, \ldots$ mit $\texttt{T}[l]\texttt{.status} \in \{$*leer*, *alt*$\}$;
    Falls $\texttt{T}[l]\texttt{.status} =$ *leer*: $\texttt{T}[l] \leftarrow (x, r,$ *voll*$)$.
    Falls $\texttt{T}[l]\texttt{.status} =$ *alt*: $(\texttt{y}, \texttt{s}) \leftarrow (\texttt{T}[l]\texttt{.key}, \texttt{T}[l]\texttt{.data})$;

Trick für Umorganisieren: (Ohne zusätzlichen Platz.)

1) `inUse` $\leftarrow$ `load`

2) Für $i = 0, 1, \ldots, m-1$ tue folgendes:
Wenn $\texttt{T}[i]\texttt{.status} =$ *voll*, setze $\texttt{T}[i]\texttt{.status} \leftarrow$ *alt*.
Wenn $\texttt{T}[i]\texttt{.status} =$ *gelöscht*, setze $\texttt{T}[i]\texttt{.status} \leftarrow$ *leer*.

3) Für $i = 0, 1, \ldots, m-1$ tue folgendes: // checke jeden Platz
Wenn $\texttt{T}[i]\texttt{.status} =$ *alt*, dann
    setze $(\texttt{x}, \texttt{r}) \leftarrow (\texttt{T}[i]\texttt{.key}, \texttt{T}[i]\texttt{.data})$;
    setze $\texttt{T}[i]\texttt{.status} \leftarrow$ *leer*;
    insert$^*(\texttt{x}, \texttt{r})$.

    Dabei ist **insert***(x,r) folgende rekursive Variante von **insert**(x,r):
    Finde erstes l in der Folge $h(x, j), j = 0, 1, 2, \ldots$ mit $\texttt{T}[l]\texttt{.status} \in \{$*leer*, *alt*$\}$;
    Falls $\texttt{T}[l]\texttt{.status} =$ *leer*: $\texttt{T}[l] \leftarrow (x, r,$ *voll*$)$.
    Falls $\texttt{T}[l]\texttt{.status} =$ *alt*: $(\texttt{y}, \texttt{s}) \leftarrow (\texttt{T}[l]\texttt{.key}, \texttt{T}[l]\texttt{.data})$;
    $\texttt{T}[l] \leftarrow (x, r,$ *voll*$)$;

Trick für Umorganisieren: (Ohne zusätzlichen Platz.)

1) `inUse` $\leftarrow$ `load`

2) Für $i = 0, 1, \ldots, m-1$ tue folgendes:
Wenn $\texttt{T}[i]\texttt{.status} =$ *voll*, setze $\texttt{T}[i]\texttt{.status} \leftarrow$ *alt*.
Wenn $\texttt{T}[i]\texttt{.status} =$ *gelöscht*, setze $\texttt{T}[i]\texttt{.status} \leftarrow$ *leer*.

3) Für $i = 0, 1, \ldots, m-1$ tue folgendes: // checke jeden Platz
Wenn $\texttt{T}[i]\texttt{.status} =$ *alt*, dann
    setze $(\texttt{x}, \texttt{r}) \leftarrow (\texttt{T}[i]\texttt{.key}, \texttt{T}[i]\texttt{.data})$;
    setze $\texttt{T}[i]\texttt{.status} \leftarrow$ *leer*;
    insert$^*(\texttt{x}, \texttt{r})$.

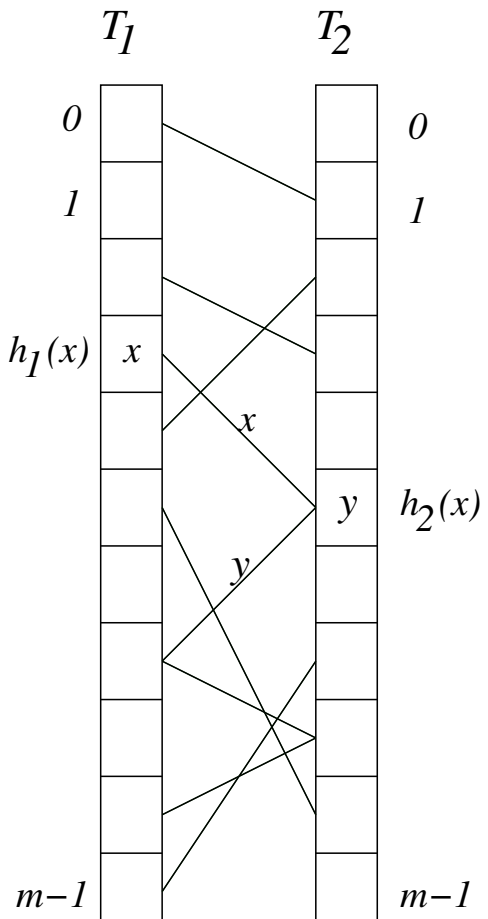
    Dabei ist **insert***(x,r) folgende rekursive Variante von **insert**(x,r):
    Finde erstes l in der Folge $h(x, j), j = 0, 1, 2, \ldots$ mit $\texttt{T}[l]\texttt{.status} \in \{$*leer*, *alt*$\}$;
    Falls $\texttt{T}[l]\texttt{.status} =$ *leer*: $\texttt{T}[l] \leftarrow (x, r,$ *voll*$)$.
    Falls $\texttt{T}[l]\texttt{.status} =$ *alt*: $(\texttt{y}, \texttt{s}) \leftarrow (\texttt{T}[l]\texttt{.key}, \texttt{T}[l]\texttt{.data})$;
    $\texttt{T}[l] \leftarrow (x, r,$ *voll*$)$;
    insert$^*(\texttt{y}, \texttt{s})$.

Vorlesungsvideo:

Cuckoo Hashing

5.5 Cuckoo Hashing [Pagh/Rodler 2001/04]

5.5 Cuckoo Hashing [Pagh/Rodler 2001/04]



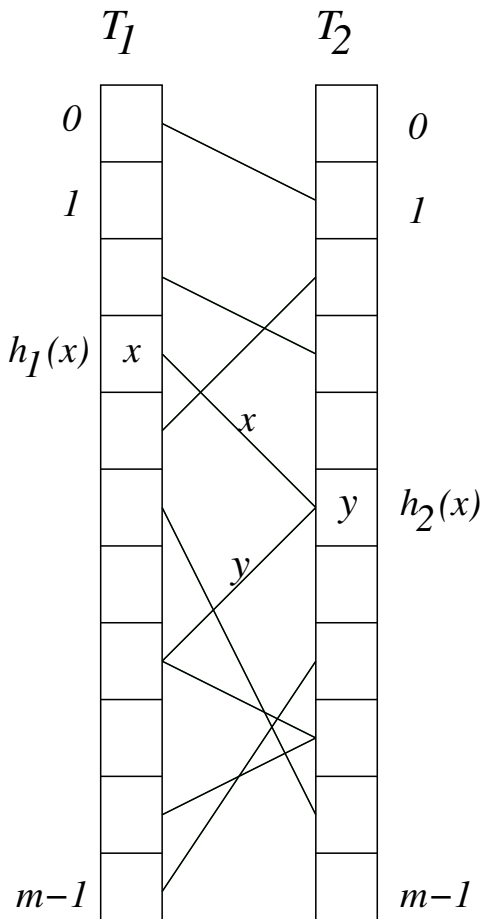
Implementierung eines dynamisches Wörterbuchs

Zwei Tabellen T_1, T_2, jeweils Größe m

Zwei Hashfunktionen h_1 und h_2

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

5.5 Cuckoo Hashing [Pagh/Rodler 2001/04]



Implementierung eines dynamisches Wörterbuchs

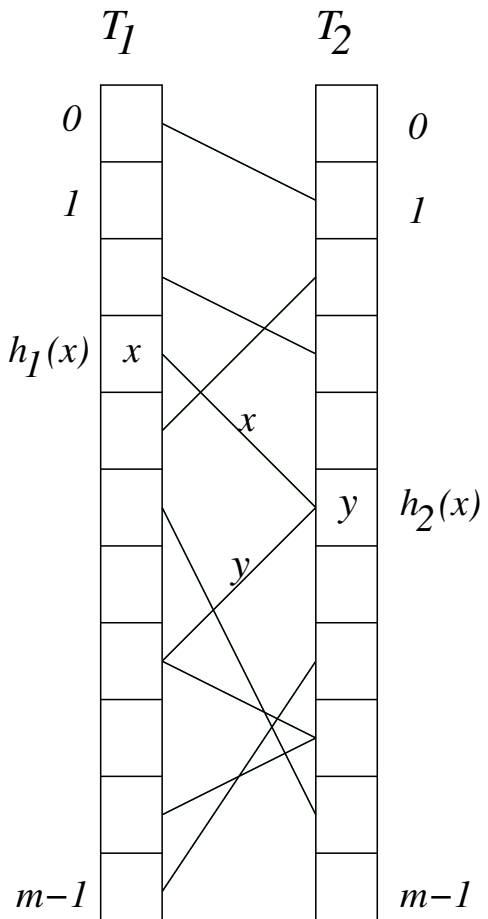
Zwei Tabellen T_1, T_2, jeweils Größe m

Zwei Hashfunktionen h_1 und h_2

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

$\Rightarrow$ **Konstante Suchzeit** garantiert.

5.5 Cuckoo Hashing [Pagh/Rodler 2001/04]



Implementierung eines dynamisches Wörterbuchs

Zwei Tabellen T_1, T_2, jeweils Größe m

Zwei Hashfunktionen h_1 und h_2

(∗) $x \in S$ sitzt 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 5.5.1

h_1, h_2 **passen zu** S, falls die Schlüssel aus S gemäß der Regel $(*)$ gespeichert werden können,

Cuckoo-Hashing

Definition 5.5.1

h_1, h_2 **passen zu** S, falls die Schlüssel aus S gemäß der Regel $(*)$ gespeichert werden können, d. h. 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 5.5.1

h_1, h_2 **passen zu** S, falls die Schlüssel aus S gemäß der Regel $(*)$ gespeichert werden können, d. h. 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.

Das Verfahren heißt **"Kuckucks-Hashing"** wegen seiner Einfügeprozedur:

Cuckoo-Hashing

Definition 5.5.1

h_1, h_2 **passen zu** S, falls die Schlüssel aus S gemäß der Regel $(*)$ gespeichert werden können, d. h. 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.

Das Verfahren heißt **"Kuckucks-Hashing"** wegen seiner 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 5.5.1

h_1, h_2 **passen zu** S, falls die Schlüssel aus S gemäß der Regel $(*)$ gespeichert werden können, d. h. 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.

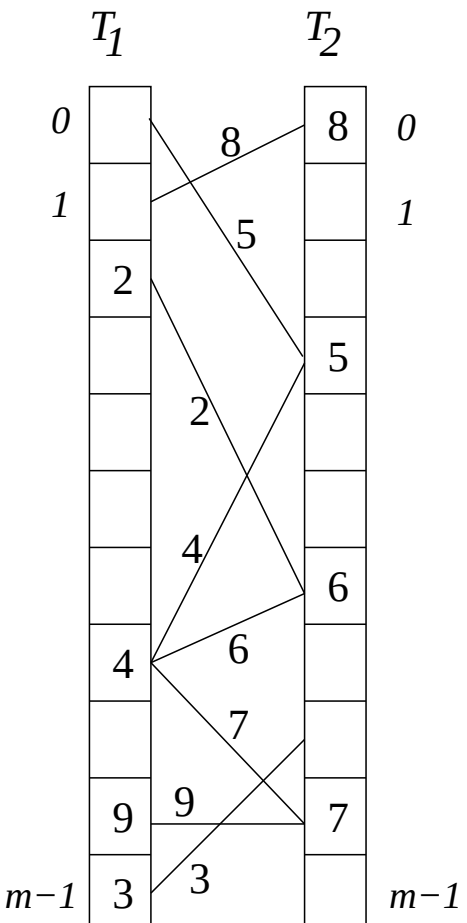
Das Verfahren heißt **"Kuckucks-Hashing"** wegen seiner Einfügeprozedur:

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,

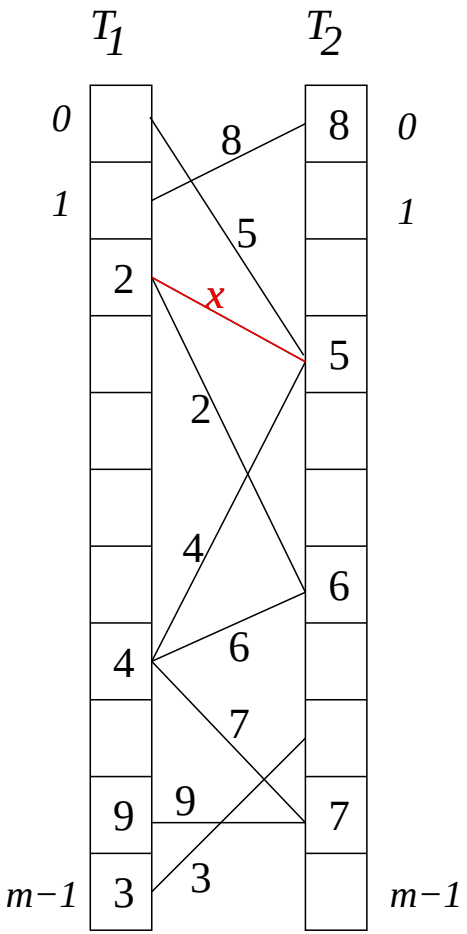
hinauswerfen (verdrängen).

Der verdrängte Schlüssel geht zu seinem alternativen Platz.

Cuckoo-Hashing

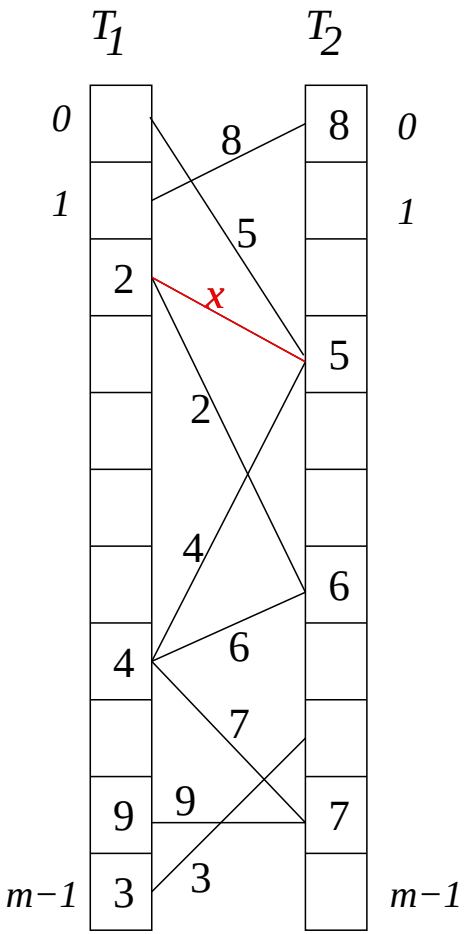


Cuckoo-Hashing



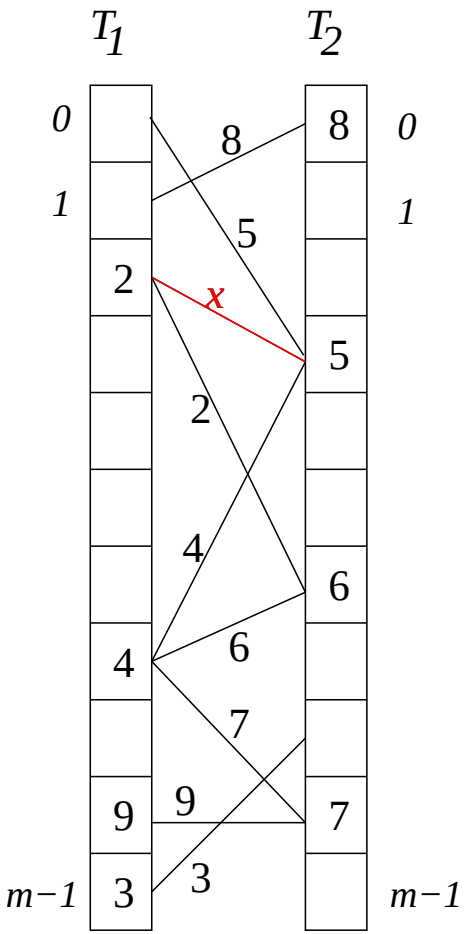
Füge x ein.

Cuckoo-Hashing



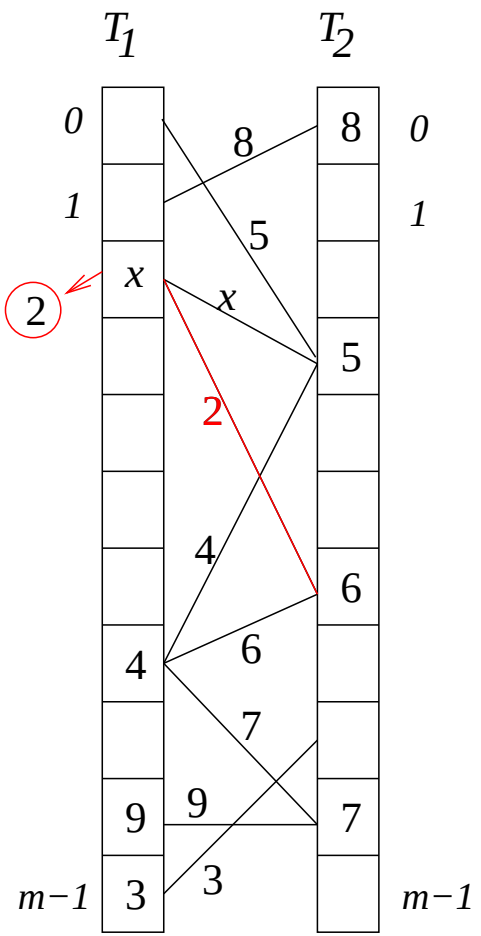
Füge x ein.
Versuche $T_1[h_1(x)]$.

Cuckoo-Hashing



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

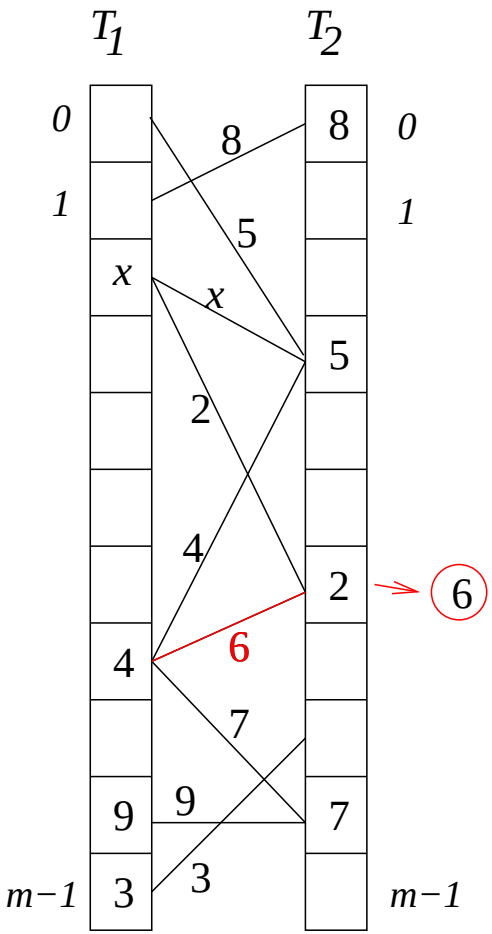
Cuckoo-Hashing



Schlüssel x in T_1 einfügen.

Schlüssel 2 aus T_1 hinauswerfen, soll nach T_2 gehen.

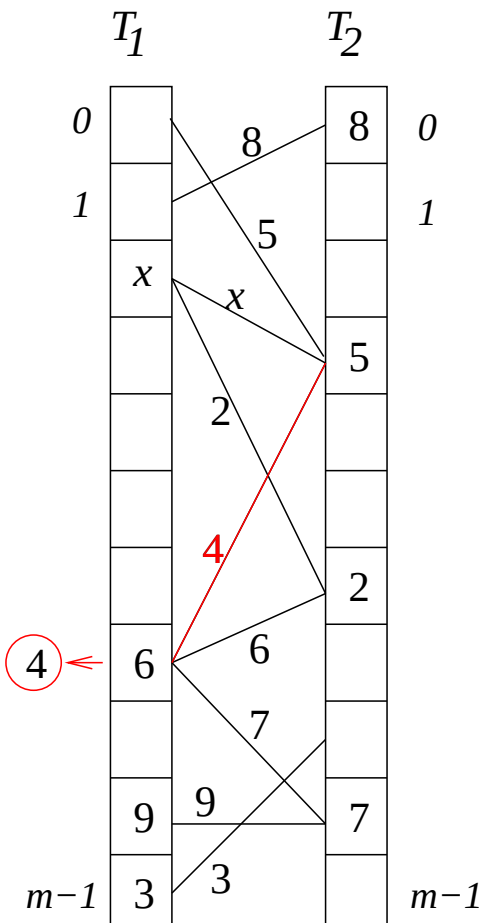
Cuckoo-Hashing



Schlüssel 2 in T_2 einfügen.

Schlüssel 6 aus T_2 hinauswerfen,
soll nach T_1 gehen.

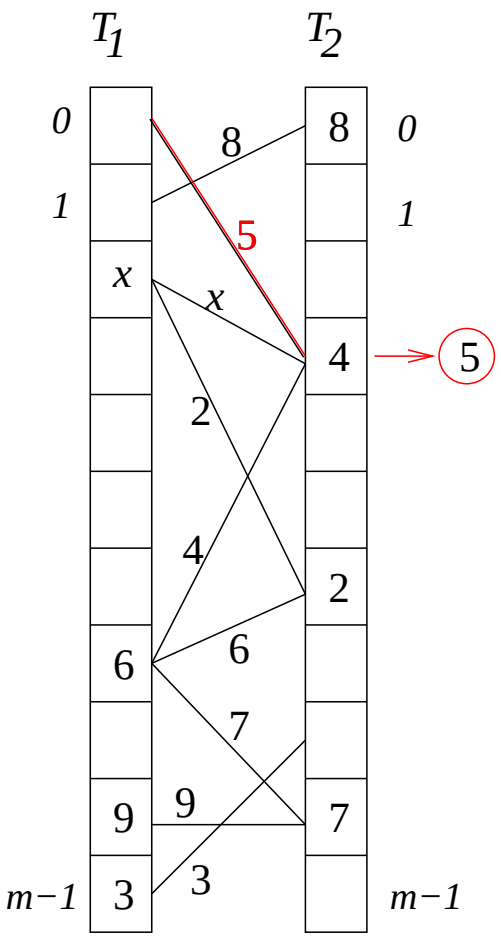
Cuckoo-Hashing



Schlüssel 6 in T_1 einfügen.

Schlüssel 4 aus T_1 hinauswerfen,
soll nach T_2 gehen.

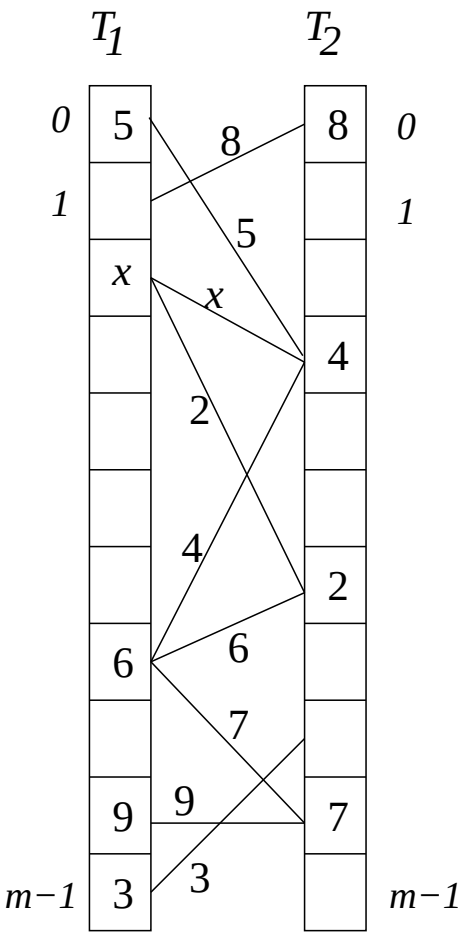
Cuckoo-Hashing



Schlüssel 4 in T_2 einfügen.

Schlüssel 5 aus T_2 hinauswerfen,
soll nach T_1 gehen.

Cuckoo-Hashing



Schlüssel 5 in T_1 einfügen.

Funktioniert, weil $T_1[h_1(5)]$ leer ist.

Fertig!

Cuckoo-Hashing: Teil-Analyse

Grundfrage: Wann funktioniert das überhaupt?

Cuckoo-Hashing: Teil-Analyse

Grundfrage: Wann funktioniert das überhaupt?

Hier zunächst: Wie lange dauert eine Einfügung?

Cuckoo-Hashing: Teil-Analyse

Grundfrage: Wann funktioniert das überhaupt?

Hier zunächst: Wie lange dauert eine Einfügung?

Wie viele Verdrängungsschritte gibt es bei Einfügung von x?

Cuckoo-Hashing: Teil-Analyse

Grundfrage: Wann funktioniert das überhaupt?

Hier zunächst: Wie lange dauert eine Einfügung?

Wie viele Verdrängungsschritte gibt es bei Einfügung von x?

Wir beobachten: Wenn die Einfügung von x zur Verdrängung von y führt, dann gibt es eine Folge von Schlüsseln $x_1, \ldots, x_k$ mit:

x verdrängt x_1

Cuckoo-Hashing: Teil-Analyse

Grundfrage: Wann funktioniert das überhaupt?

Hier zunächst: Wie lange dauert eine Einfügung?

Wie viele Verdrängungsschritte gibt es bei Einfügung von x?

Wir beobachten: Wenn die Einfügung von x zur Verdrängung von y führt, dann gibt es eine Folge von Schlüsseln $x_1, \ldots, x_k$ mit:

x verdrängt x_1, x_1 verdrängt x_2

Cuckoo-Hashing: Teil-Analyse

Grundfrage: Wann funktioniert das überhaupt?

Hier zunächst: Wie lange dauert eine Einfügung?

Wie viele Verdrängungsschritte gibt es bei Einfügung von x?

Wir beobachten: Wenn die Einfügung von x zur Verdrängung von y führt, dann gibt es eine Folge von Schlüsseln $x_1, \ldots, x_k$ mit:

x verdrängt x_1, x_1 verdrängt x_2, . . . , x_k verdrängt y.

Damit das passieren kann, muss es eine einfache „$x..y$-Kette“ $x, x_1, x_2, \ldots, x_k, y$ aus *verschiedenen* Schlüsseln geben, die über die Hashfunktionen eine Verbindung herstellt:

Damit das passieren kann, muss es eine einfache „$x..y$-Kette“ $x, x_1, x_2, \ldots, x_k, y$ aus *verschiedenen* Schlüsseln geben, die über die Hashfunktionen eine Verbindung herstellt:

$$h_1(x) = h_1(x_1),$$

Damit das passieren kann, muss es eine einfache „$x..y$-Kette“ $x, x_1, x_2, \ldots, x_k, y$ aus *verschiedenen* Schlüsseln geben, die über die Hashfunktionen eine Verbindung herstellt:

$$h_1(x) = h_1(x_1),$$
$$h_2(x_1) = h_2(x_2)$$

Damit das passieren kann, muss es eine einfache „$x..y$-Kette“ $x, x_1, x_2, \ldots, x_k, y$ aus *verschiedenen* Schlüsseln geben, die über die Hashfunktionen eine Verbindung herstellt:

$$h_1(x) = h_1(x_1),$$
$$h_2(x_1) = h_2(x_2)$$
$$h_1(x_2) = h_1(x_3)$$

Damit das passieren kann, muss es eine einfache „$x..y$-Kette“ $x, x_1, x_2, \ldots, x_k, y$ aus *verschiedenen* Schlüsseln geben, die über die Hashfunktionen eine Verbindung herstellt:

$$\begin{aligned} h_1(x) &= h_1(x_1), \\ h_2(x_1) &= h_2(x_2) \\ h_1(x_2) &= h_1(x_3) \\ &\vdots \\ h_1(x_\ell) &= h_1(y). \end{aligned}$$

Damit das passieren kann, muss es eine einfache „$x..y$-Kette“ $x, x_1, x_2, \ldots, x_k, y$ aus *verschiedenen* Schlüsseln geben, die über die Hashfunktionen eine Verbindung herstellt:

$$\begin{aligned} h_1(x) &= h_1(x_1), \\ h_2(x_1) &= h_2(x_2) \\ h_1(x_2) &= h_1(x_3) \\ &\vdots \\ h_1(x_\ell) &= h_1(y). \end{aligned}$$

(Auch möglich: Die Folge beginnt mit $h_2(x) = h_2(x_1)$ oder sie endet mit $h_2(x_\ell) = h_2(y)$.)

Wir sammeln alle in Frage kommenden y in eine (Zufalls-)Menge

Wir sammeln alle in Frage kommenden y in eine (Zufalls-)Menge

$$\boldsymbol{B_x} := \{y \in S \mid \text{es gibt eine (}\textit{einfache}\text{) } x \ldots y\text{-Kette}\}.$$

Wir sammeln alle in Frage kommenden y in eine (Zufalls-)Menge

$$\boldsymbol{B_x} := \{y \in S \mid \text{es gibt eine (}\textit{einfache}\text{) } x \ldots y\text{-Kette}\}.$$

Wir wollen die erwartete Größe $\mathbf{E}(|B_x|)$ abschätzen.

Wir sammeln alle in Frage kommenden y in eine (Zufalls-)Menge

$$\boldsymbol{B_x} := \{y \in S \mid \text{es gibt eine (}\textit{einfache}\text{) } x \ldots y\text{-Kette}\}.$$

Wir wollen die erwartete Größe $\mathbf{E}(|B_x|)$ abschätzen.

Dazu schreiben wir $|B_x| = \sum_{y \in S} X_y$ mit

$$\boldsymbol{X_y} := \begin{cases} 1 & \text{wenn } y \in B_x \\ 0 & \text{wenn } y \notin B_x. \end{cases}$$

Wir sammeln alle in Frage kommenden y in eine (Zufalls-)Menge

$$\boldsymbol{B_x} := \{y \in S \mid \text{es gibt eine (}\textit{einfache}\text{) } x \ldots y\text{-Kette}\}.$$

Wir wollen die erwartete Größe $\mathbf{E}(|B_x|)$ abschätzen.

Dazu schreiben wir $|B_x| = \sum_{y \in S} X_y$ mit

$$\boldsymbol{X_y} := \begin{cases} 1 & \text{wenn } y \in B_x \\ 0 & \text{wenn } y \notin B_x. \end{cases}$$

Dann gilt (Linearität des Erwartungswertes):

$$\mathbf{E}(|B_x|) = \sum_{y \in S} \mathbf{E}(X_y) = \sum_{y \in S} \mathbf{Pr}(\text{es gibt } x \ldots y\text{-Kette}).$$

Wir schätzen $\mathbf{E}(X_y)$ mit Hilfe der „Vereinigungsschranke“ $\mathbf{Pr}(A_1 \cup \cdots \cup A_\ell) \leq \mathbf{Pr}(A_1) + \cdots + \mathbf{Pr}(A_\ell)$ ab:

Wir schätzen $\mathbf{E}(X_y)$ mit Hilfe der „Vereinigungsschranke“ $\mathbf{Pr}(A_1 \cup \cdots \cup A_\ell) \leq \mathbf{Pr}(A_1) + \cdots + \mathbf{Pr}(A_\ell)$ ab:

$$\mathbf{E}(X_y) = \mathbf{Pr}(\exists k \geq 1\text{: es gibt einfache } x \ldots y\text{-Kette der Länge } k)$$

Wir schätzen $\mathbf{E}(X_y)$ mit Hilfe der „Vereinigungsschranke“ $\mathbf{Pr}(A_1 \cup \cdots \cup A_\ell) \leq \mathbf{Pr}(A_1) + \cdots + \mathbf{Pr}(A_\ell)$ ab:

$$\begin{aligned}\mathbf{E}(X_y) &= \mathbf{Pr}(\exists k \geq 1\text{: es gibt einfache } x \ldots y\text{-Kette der Länge } k) \\ &\leq \sum_{k \geq 1} \sum_{x_1,\ldots,x_k \in S} \mathbf{Pr}(x, x_1, \ldots, x_k, y \text{ ist } x \ldots y\text{-Kette})\end{aligned}$$

Wir schätzen $\mathbf{E}(X_y)$ mit Hilfe der „Vereinigungsschranke“ $\mathbf{Pr}(A_1 \cup \cdots \cup A_\ell) \leq \mathbf{Pr}(A_1) + \cdots + \mathbf{Pr}(A_\ell)$ ab:

$$\begin{aligned}
\mathbf{E}(X_y) &= \mathbf{Pr}(\exists k \geq 1\text{: es gibt einfache } x \ldots y\text{-Kette der Länge } k) \\
&\leq \sum_{k \geq 1} \sum_{x_1,\ldots,x_k \in S} \mathbf{Pr}(x, x_1, \ldots, x_k, y \text{ ist } x \ldots y\text{-Kette}) \\
&\leq \sum_{k \geq 1} 4n^k \frac{1}{m^{k+1}}
\end{aligned}$$

($k+1$ Hashwerte sind gleich; Faktor 4 wg. h_1/h_2-Auswahl)

Wir schätzen $\mathbf{E}(X_y)$ mit Hilfe der „Vereinigungsschranke“ $\mathbf{Pr}(A_1 \cup \cdots \cup A_\ell) \leq \mathbf{Pr}(A_1) + \cdots + \mathbf{Pr}(A_\ell)$ ab:

$$\begin{aligned}
\mathbf{E}(X_y) &= \mathbf{Pr}(\exists k \geq 1\text{: es gibt einfache } x \ldots y\text{-Kette der Länge } k) \\
&\leq \sum_{k \geq 1} \sum_{x_1,\ldots,x_k \in S} \mathbf{Pr}(x, x_1, \ldots, x_k, y \text{ ist } x \ldots y\text{-Kette}) \\
&\leq \sum_{k \geq 1} 4n^k \frac{1}{m^{k+1}}
\end{aligned}$$

($k+1$ Hashwerte sind gleich; Faktor 4 wg. h_1/h_2-Auswahl)

$$\cdots = \frac{4}{m} \cdot \sum_{k \geq 1} \left(\frac{n}{m}\right)^k$$

Wir schätzen $\mathbf{E}(X_y)$ mit Hilfe der „Vereinigungsschranke“ $\mathbf{Pr}(A_1 \cup \cdots \cup A_\ell) \leq \mathbf{Pr}(A_1) + \cdots + \mathbf{Pr}(A_\ell)$ ab:

$$\begin{aligned}\mathbf{E}(X_y) &= \mathbf{Pr}(\exists k \geq 1\text{: es gibt einfache } x \ldots y\text{-Kette der Länge } k) \\ &\leq \sum_{k \geq 1} \sum_{x_1,\ldots,x_k \in S} \mathbf{Pr}(x, x_1, \ldots, x_k, y \text{ ist } x \ldots y\text{-Kette}) \\ &\leq \sum_{k \geq 1} 4n^k \frac{1}{m^{k+1}}\end{aligned}$$

($k+1$ Hashwerte sind gleich; Faktor 4 wg. h_1/h_2-Auswahl)

$$\cdots = \frac{4}{m} \cdot \sum_{k \geq 1} \left(\frac{n}{m}\right)^k \leq \frac{4}{m} \cdot \sum_{k \geq 0} (1-\beta)^k$$

Wir schätzen $\mathbf{E}(X_y)$ mit Hilfe der „Vereinigungsschranke“ $\mathbf{Pr}(A_1 \cup \cdots \cup A_\ell) \leq \mathbf{Pr}(A_1) + \cdots + \mathbf{Pr}(A_\ell)$ ab:

$$\begin{aligned}\mathbf{E}(X_y) &= \mathbf{Pr}(\exists k \geq 1\text{: es gibt einfache } x \ldots y\text{-Kette der Länge } k) \\ &\leq \sum_{k\geq 1} \sum_{x_1,\ldots,x_k \in S} \mathbf{Pr}(x, x_1, \ldots, x_k, y \text{ ist } x \ldots y\text{-Kette}) \\ &\leq \sum_{k\geq 1} 4n^k \frac{1}{m^{k+1}}\end{aligned}$$

($k+1$ Hashwerte sind gleich; Faktor 4 wg. h_1/h_2-Auswahl)

$$\cdots = \frac{4}{m} \cdot \sum_{k\geq 1} \left(\frac{n}{m}\right)^k \leq \frac{4}{m} \cdot \sum_{k\geq 0} (1-\beta)^k = \frac{4}{m} \cdot \frac{1}{\beta},$$

falls $n/m \leq 1 - \beta$.

Wir schätzen $\mathbf{E}(X_y)$ mit Hilfe der „Vereinigungsschranke“ $\mathbf{Pr}(A_1 \cup \cdots \cup A_\ell) \leq \mathbf{Pr}(A_1) + \cdots + \mathbf{Pr}(A_\ell)$ ab:

$$\begin{aligned}\mathbf{E}(X_y) &= \mathbf{Pr}(\exists k \geq 1\text{: es gibt einfache } x \ldots y\text{-Kette der Länge } k) \\ &\leq \sum_{k\geq 1} \sum_{x_1,\ldots,x_k \in S} \mathbf{Pr}(x, x_1, \ldots, x_k, y \text{ ist } x \ldots y\text{-Kette}) \\ &\leq \sum_{k\geq 1} 4n^k \frac{1}{m^{k+1}}\end{aligned}$$

($k+1$ Hashwerte sind gleich; Faktor 4 wg. h_1/h_2-Auswahl)

$$\cdots = \frac{4}{m} \cdot \sum_{k\geq 1} \left(\frac{n}{m}\right)^k \leq \frac{4}{m} \cdot \sum_{k\geq 0} (1-\beta)^k = \frac{4}{m} \cdot \frac{1}{\beta},$$

falls $n/m \leq 1-\beta$. Damit: $\mathbf{E}(|B_x|) \leq 4n/(m\beta) \leq \frac{4(1-\beta)}{\beta} < 4/\beta$.

Bei einer erfolgreichen Einfügung wird kein Schlüssel mehr als zweimal verdrängt.

Wir erhalten also

$$\mathbf{E}(\text{Einfügezeit}) = O(\mathbf{E}(|B_x|)) = O(1/\beta).$$

Leider ist die Analyse nicht ganz vollständig. Man muss noch die Wahrscheinlichkeit abschätzen, dass h_1 und h_2 nicht zu S passen. Diese ergibt sich (mit etwas komplizierteren Überlegungen) zu $\Theta(1/n)$.

Damit kann man dann die Analyse vervollständigen: Erwartete Einfügezeit ist $O(1)$, selbst wenn man den mit Wahrscheinlichkeit $O(1/n)$ nötigen Neuaufbau mit in Rechnung stellt.

Literatur: Rasmus Pagh, Cuckoo Hashing for Undergraduates, March 27, 2006, `https://www.itu.dk/people/pagh/papers/cuckoo-undergrad.pdf`

Cuckoo-Hashing: Analyse

Beobachtung 1 (trivial):

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

Cuckoo-Hashing: Analyse

Beobachtung 1 (trivial):

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

$\Rightarrow$

h_1, h_2 passen zu S.

Cuckoo-Hashing: Analyse

Beobachtung 1 (trivial):

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

$\Rightarrow$

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

Beobachtung 1 (trivial):

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

$\Rightarrow$

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: 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

$\Rightarrow$

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

$\Rightarrow$

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

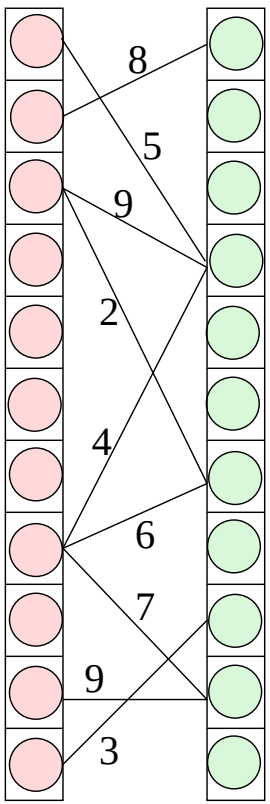
$\Rightarrow$

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?

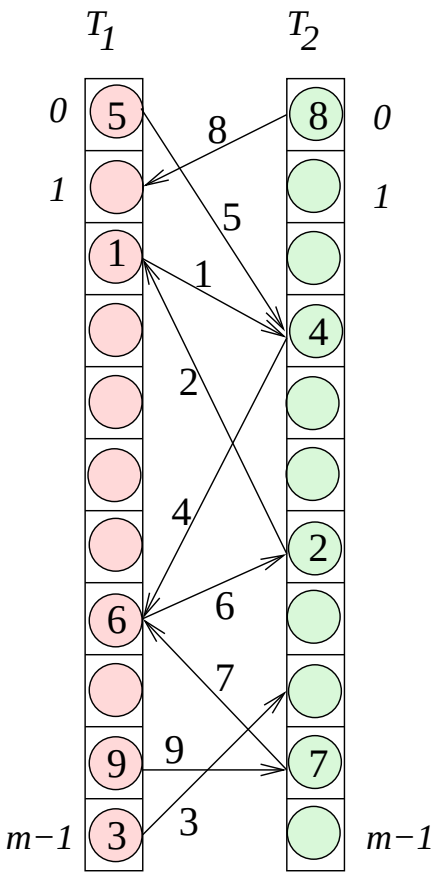
Cuckoo-Hashing: Graph



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

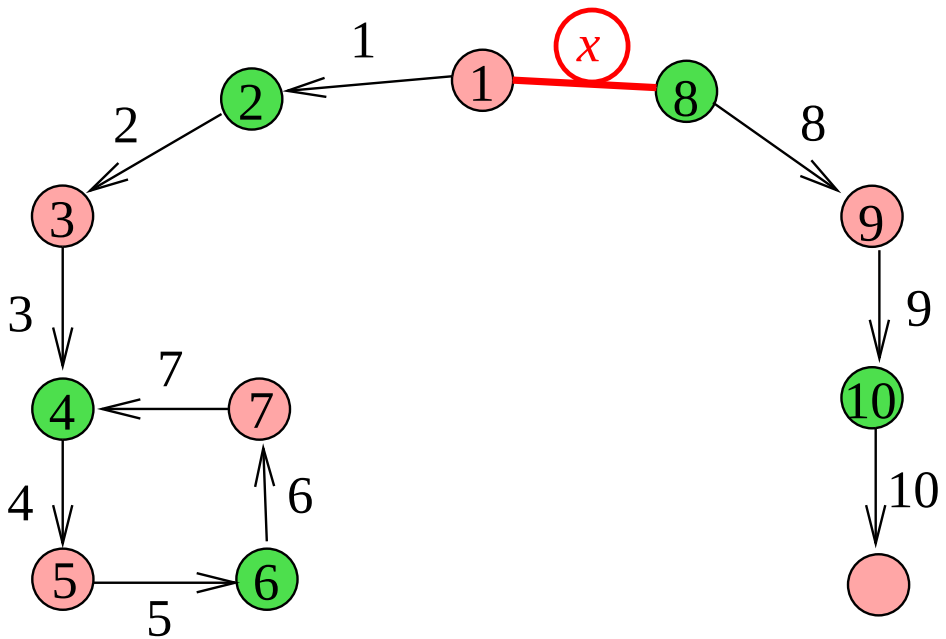
$V = W = [m]$
$E = \{(h_1(x), h_2(x)) \mid x \in S\}$

Cuckoo-Hashing: Graph



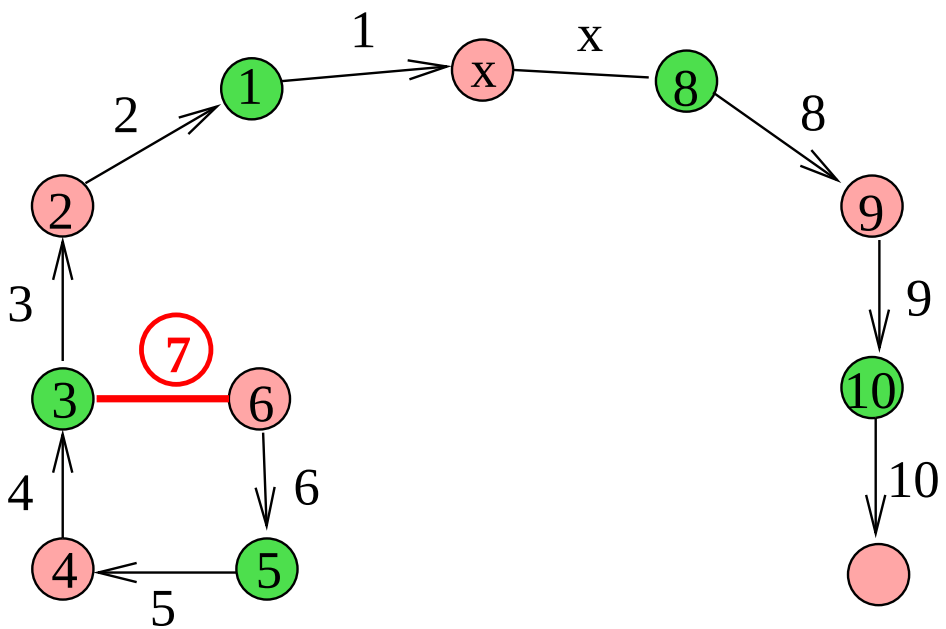
h_1, h_2 passen zu S
$\Leftrightarrow$
Man kann die Kanten von $G(S, h_1, h_2)$ so richten, dass jeder Knoten Ausgrad ≤ 1 hat.

Cuckoo-Hashing: Fehlschlag?



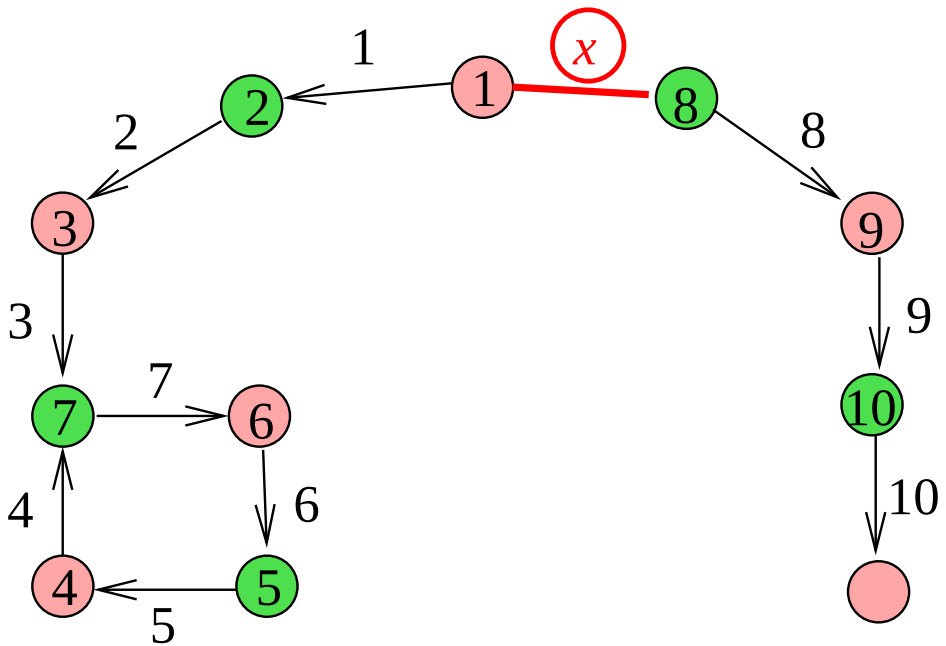
Neuer Schlüssel x.

Cuckoo-Hashing: Fehlschlag?



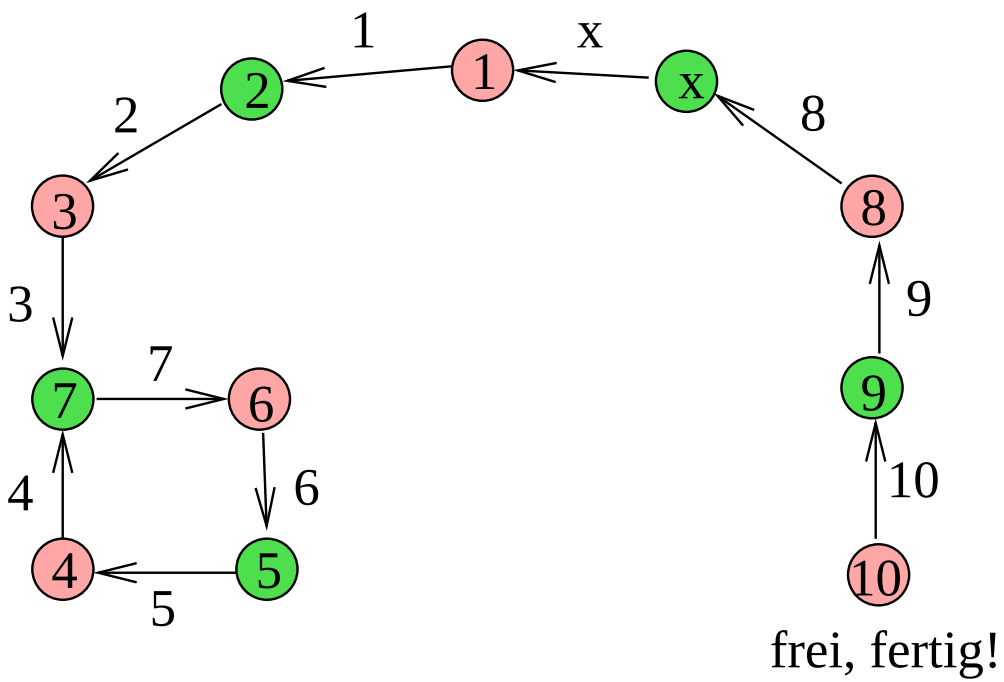
Nach 6 Runden. Schlüssel 7 ist draußen.

Cuckoo-Hashing: Fehlschlag?

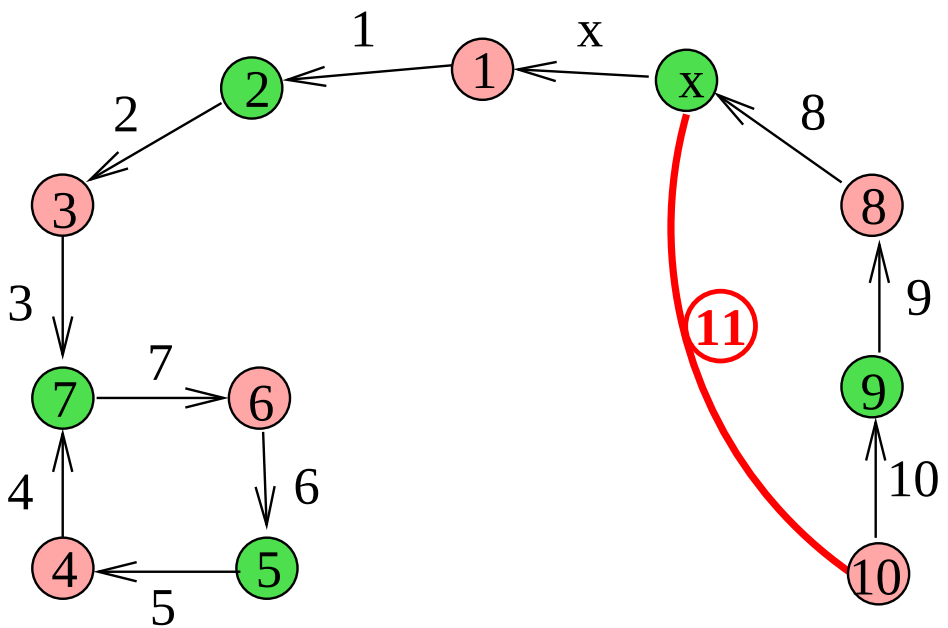


Nach weiteren 4 Runden.

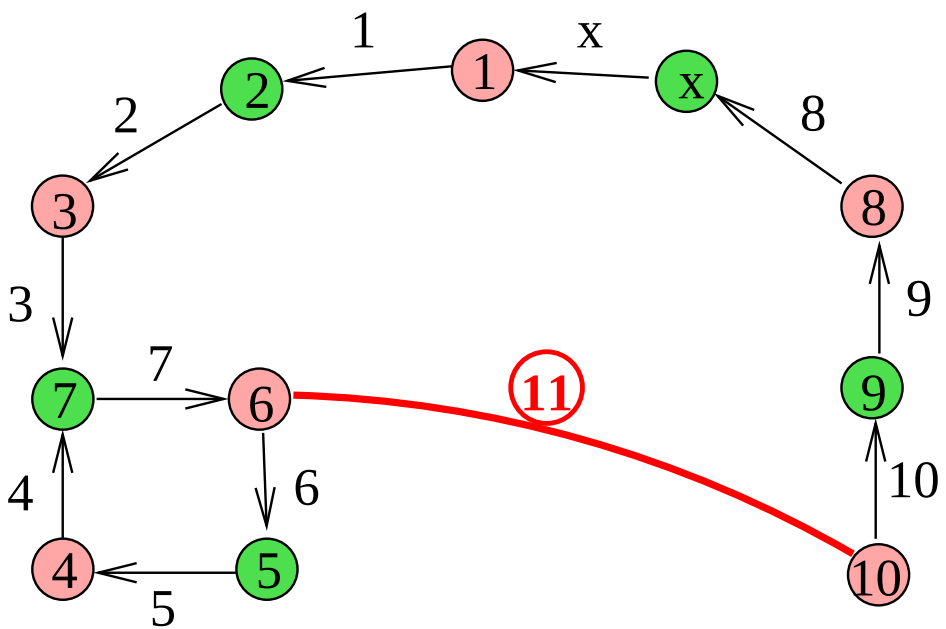
Cuckoo-Hashing: Fehlschlag?



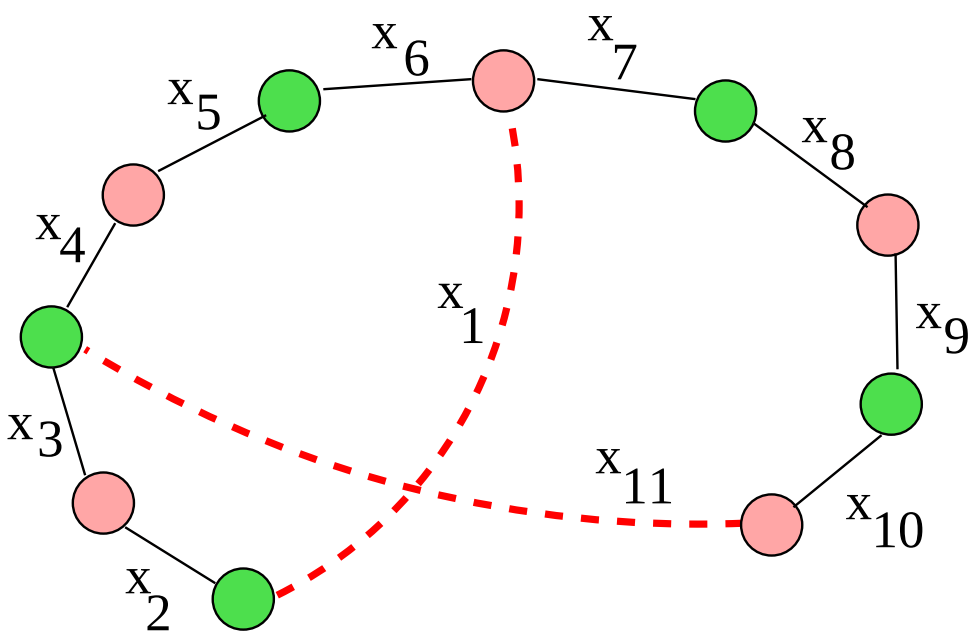
Cuckoo-Hashing: Fehlschlag!



Cuckoo-Hashing: Fehlschlag!

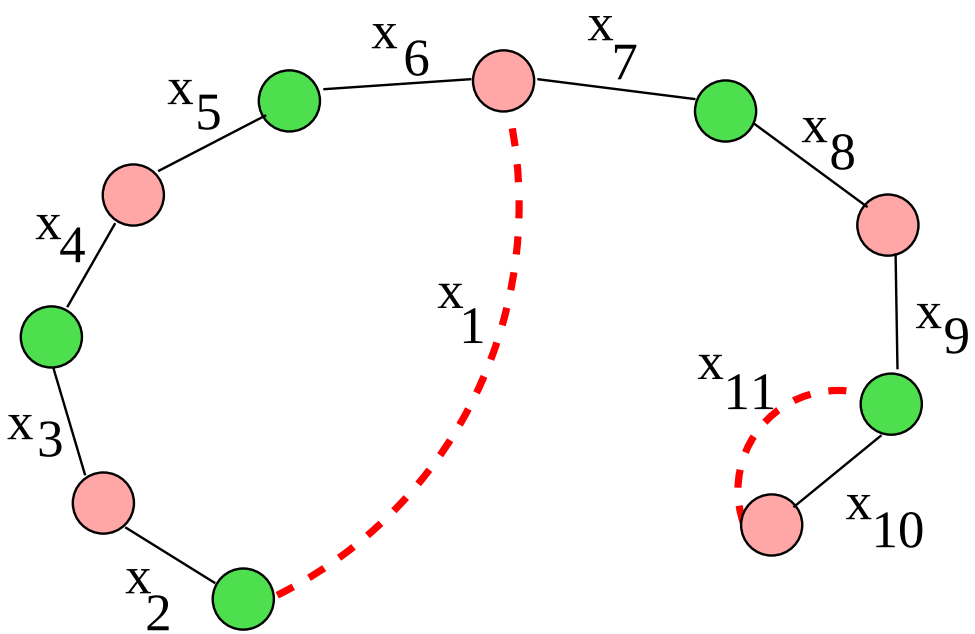


Cuckoo-Hashing: Fehlschlag



„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 $x_1, \ldots, x_k$ 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 $x_1, \ldots, x_k$ 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 $x_1, \ldots, x_k$ in S, deren Hashwerte wie folgt kollidieren (z. B., falls k ungerade):

$$h_2(x_1) = h_2(x_2), \qquad h_1(x_2) = h_1(x_3),$$

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 $x_1, \ldots, x_k$ in S, deren Hashwerte wie folgt kollidieren (z. B., falls k ungerade):

$$h_2(x_1) = h_2(x_2), \qquad h_1(x_2) = h_1(x_3),$$

$$h_2(x_3) = h_2(x_4), \qquad h_1(x_4) = h_1(x_5),$$

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 $x_1, \ldots, x_k$ in S, deren Hashwerte wie folgt kollidieren (z. B., falls k ungerade):

$$h_2(x_1) = h_2(x_2), \qquad h_1(x_2) = h_1(x_3),$$
$$h_2(x_3) = h_2(x_4), \qquad h_1(x_4) = h_1(x_5),$$
$$\vdots \qquad\qquad \vdots$$
$$h_2(x_{k-2}) = h_2(x_{k-1}), \qquad h_1(x_{k-1}) = h_1(x_k),$$

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 $x_1, \ldots, x_k$ in S, deren Hashwerte wie folgt kollidieren (z. B., falls k ungerade):

$$h_2(x_1) = h_2(x_2), \qquad h_1(x_2) = h_1(x_3),$$
$$h_2(x_3) = h_2(x_4), \qquad h_1(x_4) = h_1(x_5),$$
$$\vdots \qquad\qquad \vdots$$
$$h_2(x_{k-2}) = h_2(x_{k-1}), \qquad h_1(x_{k-1}) = h_1(x_k),$$
$$h_1(x_1) \in \{h_1(x_3), \ldots, h_1(x_k)\},$$

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 $x_1, \ldots, x_k$ in S, deren Hashwerte wie folgt kollidieren (z. B., falls k ungerade):

$$h_2(x_1) = h_2(x_2), \qquad h_1(x_2) = h_1(x_3),$$
$$h_2(x_3) = h_2(x_4), \qquad h_1(x_4) = h_1(x_5),$$
$$\vdots \qquad\qquad \vdots$$
$$h_2(x_{k-2}) = h_2(x_{k-1}), \qquad h_1(x_{k-1}) = h_1(x_k),$$

$$h_1(x_1) \in \{h_1(x_3), \ldots, h_1(x_k)\},$$
$$h_2(x_k) \in \{h_2(x_2), \ldots, h_2(x_{k-1})\}.$$

Cuckoo-Hashing: Analyse

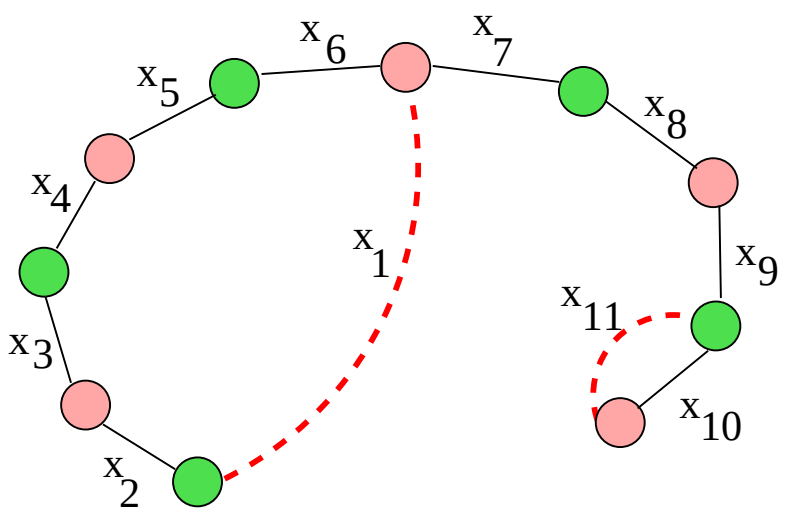
Beobachtung 2': Sogar Äquivalenz!

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!

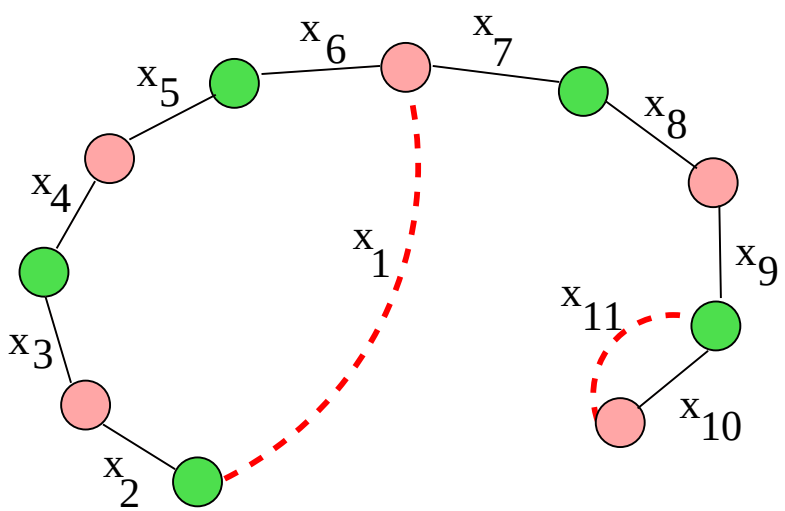
Wenn in $G(S, h_1, h_2)$ eine **Sperrstruktur** existiert,
dann passen h_1, h_2 nicht zu S.



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:

$$\mathbf{Pr}(h_1, h_2 \text{ passen nicht zu } S)$$

$$\leq$$

Cuckoo-Hashing: Analyse

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

$$\mathbf{Pr}(h_1, h_2 \text{ passen nicht zu } S)$$

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

Cuckoo-Hashing: Analyse

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

$$\textbf{Pr}(h_1, h_2 \text{ passen nicht zu } S)$$

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

Cuckoo-Hashing: Analyse

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

$$\textbf{Pr}(h_1, h_2 \text{ passen nicht zu } S)$$

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

Cuckoo-Hashing: Analyse

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

$$\mathbf{Pr}(h_1, h_2 \text{ passen nicht zu } S)$$

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

Cuckoo-Hashing: Analyse

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

$$\begin{aligned}
&\textbf{Pr}(h_1, h_2 \text{ passen nicht zu } S) \\
&\qquad \leq \sum_{3 \leq k \leq n} n^k \cdot 2 \cdot \left(\frac{1}{m}\right)^{k-1} \cdot \left(\frac{k}{2m}\right)^2 \\
&\qquad <
\end{aligned}$$

Cuckoo-Hashing: Analyse

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

$$\begin{aligned}
&\textbf{Pr}(h_1, h_2 \text{ passen nicht zu } S) \\
&\leq \sum_{3 \leq k \leq n} n^k \cdot 2 \cdot \left(\frac{1}{m}\right)^{k-1} \cdot \left(\frac{k}{2m}\right)^2 \\
&< \frac{1}{2m} \cdot \sum_{k \geq 3}
\end{aligned}$$

Cuckoo-Hashing: Analyse

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

$$\begin{aligned}
&\textbf{Pr}(h_1, h_2 \text{ passen nicht zu } S) \\
&\leq \sum_{3 \leq k \leq n} n^k \cdot 2 \cdot \left(\frac{1}{m}\right)^{k-1} \cdot \left(\frac{k}{2m}\right)^2 \\
&< \frac{1}{2m} \cdot \sum_{k \geq 3} k^2 \cdot \left(\frac{n}{m}\right)^k
\end{aligned}$$

Cuckoo-Hashing: Analyse

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

$$\begin{aligned}
&\textbf{Pr}(h_1, h_2 \text{ passen nicht zu } S) \\
&\qquad \leq \sum_{3 \leq k \leq n} n^k \cdot 2 \cdot \left(\frac{1}{m}\right)^{k-1} \cdot \left(\frac{k}{2m}\right)^2 \\
&\qquad < \frac{1}{2m} \cdot \sum_{k \geq 3} k^2 \cdot \left(\frac{n}{m}\right)^k \\
&\qquad \leq
\end{aligned}$$

Cuckoo-Hashing: Analyse

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

$$\begin{aligned}
&\textbf{Pr}(h_1, h_2 \text{ passen nicht zu } S) \\
&\leq \sum_{3 \leq k \leq n} n^k \cdot 2 \cdot \left(\frac{1}{m}\right)^{k-1} \cdot \left(\frac{k}{2m}\right)^2 \\
&< \frac{1}{2m} \cdot \sum_{k \geq 3} k^2 \cdot \left(\frac{n}{m}\right)^k \\
&\leq \frac{1}{2m} \cdot \sum_{k \geq 3}
\end{aligned}$$

Cuckoo-Hashing: Analyse

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

$$\begin{aligned}
&\textbf{Pr}(h_1, h_2 \text{ passen nicht zu } S) \\
&\leq \sum_{3 \leq k \leq n} n^k \cdot 2 \cdot \left(\frac{1}{m}\right)^{k-1} \cdot \left(\frac{k}{2m}\right)^2 \\
&< \frac{1}{2m} \cdot \sum_{k \geq 3} k^2 \cdot \left(\frac{n}{m}\right)^k \\
&\leq \frac{1}{2m} \cdot \sum_{k \geq 3} \cdot k^2 \cdot (1-\beta)^k
\end{aligned}$$

Cuckoo-Hashing: Analyse

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

$$\begin{aligned}
&\textbf{Pr}(h_1, h_2 \text{ passen nicht zu } S) \\
&\qquad \leq \sum_{3 \leq k \leq n} n^k \cdot 2 \cdot \left(\frac{1}{m}\right)^{k-1} \cdot \left(\frac{k}{2m}\right)^2 \\
&\qquad < \frac{1}{2m} \cdot \sum_{k \geq 3} k^2 \cdot \left(\frac{n}{m}\right)^k \\
&\qquad \leq \frac{1}{2m} \cdot \sum_{k \geq 3} \cdot k^2 \cdot (1-\beta)^k = O\left(\frac{1}{\beta^3 \cdot m}\right).
\end{aligned}$$

Cuckoo-Hashing: Analyse

(Es gibt $\leq n^k$ Folgen $(x_1, \ldots, x_k)$ 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) abzubrechen.

Cuckoo-Hashing: Analyse

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

Lasse Einfügezyklus für **maximal** $C \log n$ Schritte laufen,
C eine Konstante.
Danach: „Rehash“ (neue Hashfunktion, Neuaufbau).

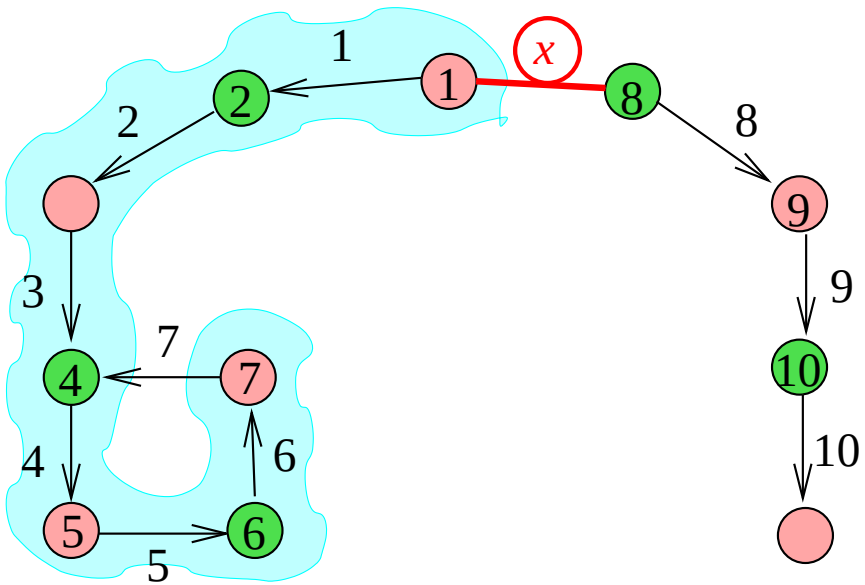
Cuckoo-Hashing: Analyse

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

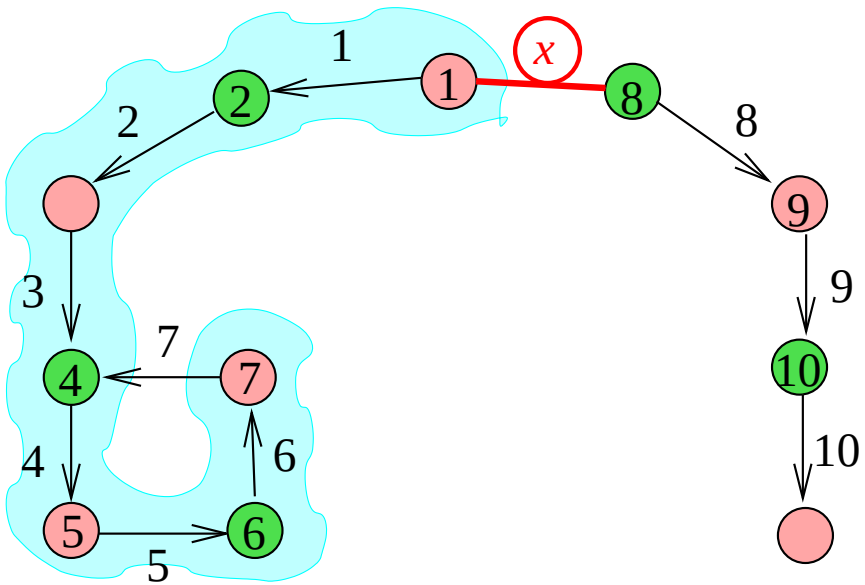
Lasse Einfügezyklus für **maximal** $C \log n$ Schritte laufen,
C eine Konstante.
Danach: „Rehash“ (neue Hashfunktion, Neuaufbau).

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

$\mathbf{Pr}(\geq t$ Verdrängungsschritte$)$

Cuckoo-Hashing: Analyse

$$
\begin{aligned}
&\textbf{Pr}(\geq t \text{ Verdrängungsschritte}) \\
&\quad \leq 2 \cdot n^{\lceil (t-1)/3 \rceil} \left(\frac{1}{m}\right)^{\lceil (t-1)/3 \rceil} \\
&\quad \leq 2 \cdot (1-\beta)^{(t-1)/3} \\
&\quad \leq 2^{c-t/d} \text{ , für Konstante } c, d.
\end{aligned}
$$

Cuckoo-Hashing: Analyse

$$
\begin{aligned}
&\textbf{Pr}(\geq t \text{ Verdrängungsschritte}) \\
&\quad \leq 2 \cdot n^{\lceil (t-1)/3 \rceil} \left(\frac{1}{m}\right)^{\lceil (t-1)/3 \rceil} \\
&\quad \leq 2 \cdot (1-\beta)^{(t-1)/3} \\
&\quad \leq 2^{c-t/d} \text{ , für Konstante } c, d.
\end{aligned}
$$

$\Rightarrow$ **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

$$
\begin{aligned}
&\textbf{Pr}(\geq t \text{ Verdrängungsschritte}) \\
&\quad \leq 2 \cdot n^{\lceil (t-1)/3 \rceil} \left(\frac{1}{m}\right)^{\lceil (t-1)/3 \rceil} \\
&\quad \leq 2 \cdot (1-\beta)^{(t-1)/3} \\
&\quad \leq 2^{c-t/d} \text{ , für Konstante } c, d.
\end{aligned}
$$

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

Cuckoo-Hashing: Analyse

Einfügung von x kostet:

$$\mathbf{E}(\text{Anzahl Verdrängungsschritte})$$

* Standardformel für Erwartungswerte.

Cuckoo-Hashing: Analyse

Einfügung von x kostet:

$$\begin{aligned}
&\mathbf{E}(\text{Anzahl Verdrängungsschritte}) \\
&\overset{*}{\leq} \sum_{t \geq 1} \mathbf{Pr}(\geq t \text{ Verdrängungsschritte})
\end{aligned}$$

$*$ Standardformel für Erwartungswerte.

Cuckoo-Hashing: Analyse

Einfügung von x kostet:

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

$*$ Standardformel für Erwartungswerte.

Cuckoo-Hashing: Zusammenfassung

Mitteilung 5.5.2 Vor.: **(UF*)** Hashfunktionen verteilen Schlüssel **rein zufällig**. Wenn man Cuckoo-Hashing mit n Schlüsseln in zwei Tabellen mit je m Plätzen durchführt, wobei $\boldsymbol{n/m \leq 1 - \beta}$ (essenziell!), dann passen h_1, h_2 mit Wahrscheinlichkeit $1 - O(1/m)$ zur Schlüsselmenge S.

Cuckoo-Hashing: Zusammenfassung

Mitteilung 5.5.2 Vor.: **(UF*)** Hashfunktionen verteilen Schlüssel **rein zufällig**. Wenn man Cuckoo-Hashing mit n Schlüsseln in zwei Tabellen mit je m Plätzen durchführt, wobei $\boldsymbol{n/m \leq 1 - \beta}$ (essenziell!), dann passen h_1, h_2 mit Wahrscheinlichkeit $1 - O(1/m)$ zur Schlüsselmenge S.

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

Cuckoo-Hashing: Zusammenfassung

Mitteilung 5.5.2 Vor.: **(UF*)** Hashfunktionen verteilen Schlüssel **rein zufällig**. Wenn man Cuckoo-Hashing mit n Schlüsseln in zwei Tabellen mit je m Plätzen durchführt, wobei $\boldsymbol{n/m \leq 1 - \beta}$ (essenziell!), dann passen h_1, h_2 mit Wahrscheinlichkeit $1 - O(1/m)$ zur Schlüsselmenge S.

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

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

Mitteilung 5.5.2 Vor.: **(UF*)** Hashfunktionen verteilen Schlüssel **rein zufällig**. Wenn man Cuckoo-Hashing mit n Schlüsseln in zwei Tabellen mit je m Plätzen durchführt, wobei $\boldsymbol{n/m \leq 1 - \beta}$ (essenziell!), dann passen h_1, h_2 mit Wahrscheinlichkeit $1 - O(1/m)$ zur Schlüsselmenge S.

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

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

Mitteilung 5.5.2 Vor.: **(UF*)** Hashfunktionen verteilen Schlüssel **rein zufällig**. Wenn man Cuckoo-Hashing mit n Schlüsseln in zwei Tabellen mit je m Plätzen durchführt, wobei $\boldsymbol{n/m \leq 1 - \beta}$ (essenziell!), dann passen h_1, h_2 mit Wahrscheinlichkeit $1 - O(1/m)$ zur Schlüsselmenge S.

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

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: $O(1)$.

Literatur: [Rasmus Pagh, Cuckoo Hashing for Undergraduates],
`https://www.itu.dk/people/pagh/papers/cuckoo-undergrad.pdf`

Nachteil: Auslastung der Tabelle **unter 50%**.

Cuckoo-Hashing: Zusammenfassung

Mitteilung 5.5.2 Vor.: **(UF*)** Hashfunktionen verteilen Schlüssel **rein zufällig**. Wenn man Cuckoo-Hashing mit n Schlüsseln in zwei Tabellen mit je m Plätzen durchführt, wobei $\boldsymbol{n/m \leq 1 - \beta}$ (essenziell!), dann passen h_1, h_2 mit Wahrscheinlichkeit $1 - O(1/m)$ zur Schlüsselmenge S.

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

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: $O(1)$.

Literatur: [Rasmus Pagh, Cuckoo Hashing for Undergraduates],
`https://www.itu.dk/people/pagh/papers/cuckoo-undergrad.pdf`

Nachteil: Auslastung der Tabelle **unter 50%**.
Abhilfe: Mehr als 2 Hashfunktionen oder größere Behälter. (Details in Spezialvorlesungen.)

Cuckoo-Hashing: Zusammenfassung

Mitteilung 5.5.2 Vor.: **(UF*)** Hashfunktionen verteilen Schlüssel **rein zufällig**. Wenn man Cuckoo-Hashing mit n Schlüsseln in zwei Tabellen mit je m Plätzen durchführt, wobei $\boldsymbol{n/m \leq 1 - \beta}$ (essenziell!), dann passen h_1, h_2 mit Wahrscheinlichkeit $1 - O(1/m)$ zur Schlüsselmenge S.

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

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: $O(1)$.

Literatur: [Rasmus Pagh, Cuckoo Hashing for Undergraduates],
`https://www.itu.dk/people/pagh/papers/cuckoo-undergrad.pdf`

Nachteil: Auslastung der Tabelle **unter 50%**.
Abhilfe: Mehr als 2 Hashfunktionen oder größere Behälter. (Details in Spezialvorlesungen.)
Die verfeinerten Verfahren sind auch **praktisch** sehr effizient (auch mit Tabellenhashing).

Multiplikative/lineare Hashfunktionen nicht benutzen!