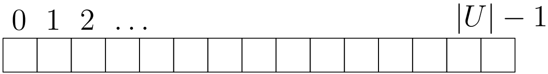
Hashtabelle: Und noch eine Datenstruktur für dynamische Mengen ...

Sei U das Universum, aus dem die Schlüssel kommen.

Speichere Daten in Feld der Länge $|U|$.

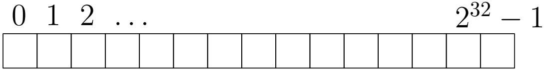


4.3 Hashing

Hashtabelle: Und noch eine Datenstruktur für dynamische Mengen ...

Sei U das Universum, aus dem die Schlüssel kommen.

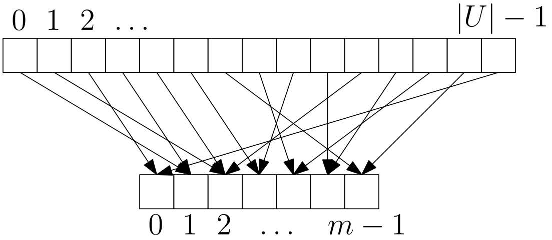
Speichere Daten in Feld der Länge $|U|$. **Nicht praktikabel!**



Hashtabelle: Und noch eine Datenstruktur für dynamische Mengen ...

Sei U das Universum, aus dem die Schlüssel kommen.

Speichere Daten in Feld der Länge $|U|$. **Nicht praktikabel!**



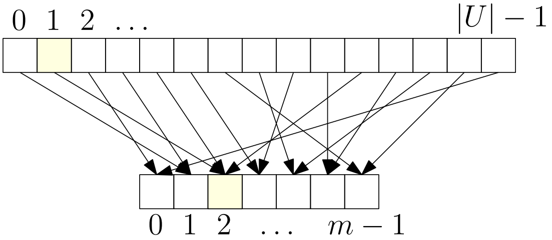
Hashfunktion $h\colon U \to \{0, 1, \ldots, m-1\}$ für $m \ll |U|$

Nutze Feld der Länge m. Speichere Element mit Schlüssel k an der Position $h(k)$.

Hashtabelle: Und noch eine Datenstruktur für dynamische Mengen ...

Sei U das Universum, aus dem die Schlüssel kommen.

Speichere Daten in Feld der Länge $|U|$. **Nicht praktikabel!**



Hashfunktion $h\colon U \to \{0, 1, \ldots, m-1\}$ für $m \ll |U|$

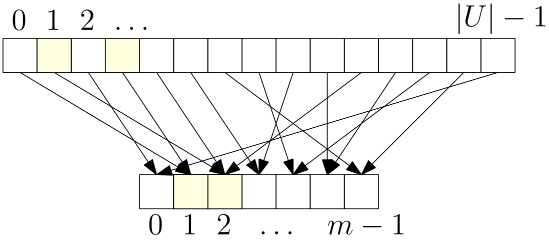
Nutze Feld der Länge m. Speichere Element mit Schlüssel k an der Position $h(k)$.

4.3 Hashing

Hashtabelle: Und noch eine Datenstruktur für dynamische Mengen . . .

Sei U das Universum, aus dem die Schlüssel kommen.

Speichere Daten in Feld der Länge $|U|$. **Nicht praktikabel!**



Hashfunktion $h\colon U \to \{0, 1, \ldots, m-1\}$ für $m \ll |U|$

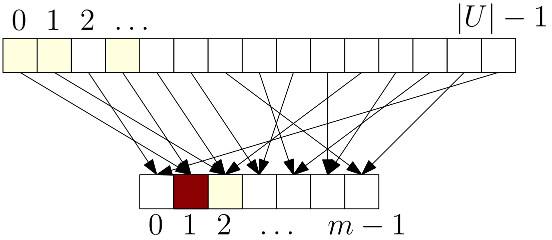
Nutze Feld der Länge m. Speichere Element mit Schlüssel k an der Position $h(k)$.

4.3 Hashing

Hashtabelle: Und noch eine Datenstruktur für dynamische Mengen ...

Sei U das Universum, aus dem die Schlüssel kommen.

Speichere Daten in Feld der Länge $|U|$. **Nicht praktikabel!**



Hashfunktion $h\colon U \to \{0, 1, \ldots, m-1\}$ für $m \ll |U|$

Nutze Feld der Länge m. Speichere Element mit Schlüssel k an der Position $h(k)$.

Problem: Kollisionen können auftreten. Wir benötigen Strategien, um damit umzugehen.

4.3.1 Hashfunktionen

Bei **guter Hashfunktion** ist $h^{-1}(i) = \{k \in U \mid h(k) = i\}$ für jedes i in etwa gleich groß.

4.3.1 Hashfunktionen

Bei **guter Hashfunktion** ist $h^{-1}(i) = \{k \in U \mid h(k) = i\}$ für jedes i in etwa gleich groß.

Theoretisch sind alle solche Hashfunktionen gleich gut, **praktisch nicht** (z. B. erster Buchstabe einer Zeichenkette als Hashfunktion).

4.3.1 Hashfunktionen

Bei **guter Hashfunktion** ist $h^{-1}(i) = \{k \in U \mid h(k) = i\}$ für jedes i in etwa gleich groß.

Theoretisch sind alle solche Hashfunktionen gleich gut, **praktisch nicht** (z. B. erster Buchstabe einer Zeichenkette als Hashfunktion).

Methoden für $U = \mathbb{N}$:

- **Divisionsmethode**: $h(k) = k \bmod m$
  m sollte keine Zweierpotenz sein. m typischerweise Primzahl.

4.3.1 Hashfunktionen

Bei **guter Hashfunktion** ist $h^{-1}(i) = \{k \in U \mid h(k) = i\}$ für jedes i in etwa gleich groß.

Theoretisch sind alle solche Hashfunktionen gleich gut, **praktisch nicht** (z. B. erster Buchstabe einer Zeichenkette als Hashfunktion).

Methoden für $U = \mathbb{N}$:

- **Divisionsmethode**: $h(k) = k \bmod m$
  m sollte keine Zweierpotenz sein. m typischerweise Primzahl.
- **Multiplikationsmethode**: $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$ für Konstante $A \in (0, 1)$.
  Die Wahl $A \approx \frac{\sqrt{5}-1}{2} = 0{,}61803\ldots$ hat sich als gut herausgestellt.

4.3.1 Hashfunktionen

Bei **guter Hashfunktion** ist $h^{-1}(i) = \{k \in U \mid h(k) = i\}$ für jedes i in etwa gleich groß.

Theoretisch sind alle solche Hashfunktionen gleich gut, **praktisch nicht** (z. B. erster Buchstabe einer Zeichenkette als Hashfunktion).

Methoden für $U = \mathbb{N}$:

- **Divisionsmethode**: $h(k) = k \bmod m$
  m sollte keine Zweierpotenz sein. m typischerweise Primzahl.
- **Multiplikationsmethode**: $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$ für Konstante $A \in (0,1)$.
  Die Wahl $A \approx \frac{\sqrt{5}-1}{2} = 0{,}61803\ldots$ hat sich als gut herausgestellt.

In Java hat jede Klasse Methode „**int** hashCode()“.

4.3.1 Hashfunktionen

Bei **guter Hashfunktion** ist $h^{-1}(i) = \{k \in U \mid h(k) = i\}$ für jedes i in etwa gleich groß.

Theoretisch sind alle solche Hashfunktionen gleich gut, **praktisch nicht** (z. B. erster Buchstabe einer Zeichenkette als Hashfunktion).

Methoden für $U = \mathbb{N}$:

- **Divisionsmethode**: $h(k) = k \bmod m$

  m sollte keine Zweierpotenz sein. m typischerweise Primzahl.

- **Multiplikationsmethode**: $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$ für Konstante $A \in (0, 1)$.

  Die Wahl $A \approx \frac{\sqrt{5}-1}{2} = 0{,}61803\ldots$ hat sich als gut herausgestellt.

In Java hat jede Klasse Methode „**int** hashCode()“.
Bei String berechnet diese $\sum_{i=0}^{\ell-1} s_i \cdot 31^i$ für Zeichenkette $s_{\ell-1} s_{\ell-2} \ldots s_1 s_0$.

4.3.1 Hashfunktionen

Bei **guter Hashfunktion** ist $h^{-1}(i) = \{k \in U \mid h(k) = i\}$ für jedes i in etwa gleich groß.

Theoretisch sind alle solche Hashfunktionen gleich gut, **praktisch nicht** (z. B. erster Buchstabe einer Zeichenkette als Hashfunktion).

Methoden für $U = \mathbb{N}$:

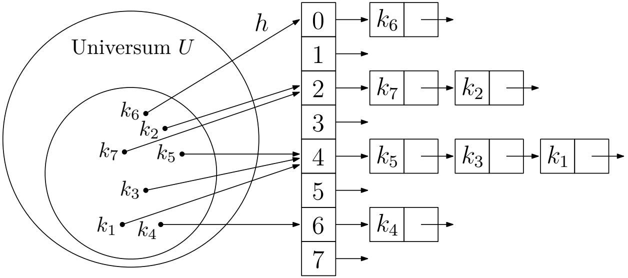
- **Divisionsmethode**: $h(k) = k \bmod m$
  m sollte keine Zweierpotenz sein. m typischerweise Primzahl.
- **Multiplikationsmethode**: $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$ für Konstante $A \in (0, 1)$.
  Die Wahl $A \approx \frac{\sqrt{5}-1}{2} = 0{,}61803\ldots$ hat sich als gut herausgestellt.

In Java hat jede Klasse Methode „**int** hashCode()“.
Bei String berechnet diese $\sum_{i=0}^{\ell-1} s_i \cdot 31^i$ für Zeichenkette $s_{\ell-1} s_{\ell-2} \ldots s_1 s_0$.

Annahme: $h(k)$ kann in konstanter Zeit berechnet werden.

Hashing mit verketteten Listen:

Lege Feld $T[0 \ldots m-1]$ an. Dabei sei jedes $T[i]$ **Zeiger auf verkettete Liste**.



Hashing mit verketteten Listen:

Lege Feld $T[0 \ldots m-1]$ an. Dabei sei jedes $T[i]$ **Zeiger auf verkettete Liste**.

- $\textsc{Search}(k)$:

  Gib das Element mit dem Schlüssel k in der Liste $T[h(k)]$ zurück, sofern es existiert.
- $\textsc{Insert}(x)$:

  Füge das Element x an den Anfang der Liste $T[h(x.\text{key})]$ ein.
- $\textsc{Delete}(k)$:

  Lösche das Element mit dem Schlüssel k aus der Liste $T[h(k)]$, sofern es existiert.

4.3.2 Hashing mit verketteten Listen

Hashing mit verketteten Listen:

Lege Feld $T[0 \ldots m-1]$ an. Dabei sei jedes $T[i]$ **Zeiger auf verkettete Liste**.

- $\textsc{Search}(k)$:
  Gib das Element mit dem Schlüssel k in der Liste $T[h(k)]$ zurück, sofern es existiert.
- $\textsc{Insert}(x)$:
  Füge das Element x an den Anfang der Liste $T[h(x.\mathrm{key})]$ ein.
- $\textsc{Delete}(k)$:
  Lösche das Element mit dem Schlüssel k aus der Liste $T[h(k)]$, sofern es existiert.

Laufzeit: im Worst Case $O(n)$

4.3.2 Hashing mit verketteten Listen

Annahme: h bildet jeden Schlüssel **uniform zufällig** und **unabhängig** auf ein Element aus $\{0, 1, \ldots, m-1\}$ ab (**uniformes Hashing**).

4.3.2 Hashing mit verketteten Listen

Annahme: h bildet jeden Schlüssel **uniform zufällig** und **unabhängig** auf ein Element aus $\{0, 1, \ldots, m-1\}$ ab (**uniformes Hashing**).

Betrachte Hashtabelle, in die die Schlüssel $k_1, \ldots, k_n$ in dieser Reihenfolge eingefügt wurden. Sei $\alpha = n/m$ der **Auslastungsfaktor**.

4.3.2 Hashing mit verketteten Listen

Annahme: h bildet jeden Schlüssel **uniform zufällig** und **unabhängig** auf ein Element aus $\{0, 1, \ldots, m-1\}$ ab (**uniformes Hashing**).

Betrachte Hashtabelle, in die die Schlüssel $k_1, \ldots, k_n$ in dieser Reihenfolge eingefügt wurden. Sei $\alpha = n/m$ der **Auslastungsfaktor**.

Theorem 4.11

Unter der Annahme des uniformen Hashings benötigt eine erfolglose Suche nach einem Schlüssel, der sich nicht in der Hashtabelle befindet, im Erwartungswert eine Laufzeit von $\Theta(1 + \alpha)$.

4.3.2 Hashing mit verketteten Listen

Beweis: Suche nach k mit $i = h(k)$. Uns interessiert $E[|T[i]|]$.

4.3.2 Hashing mit verketteten Listen

Beweis: Suche nach k mit $i = h(k)$. Uns interessiert $E[|T[i]|]$.

Für jedes $j \in \{1, \ldots, n\}$ definiere Zufallsvariable X_j:

$$X_j = \begin{cases} 1 & \text{falls } h(k_j) = i, \\ 0 & \text{falls } h(k_j) \neq i. \end{cases}$$

Beweis: Suche nach k mit $i = h(k)$. Uns interessiert $E[|T[i]|]$.

Für jedes $j \in \{1, \ldots, n\}$ definiere Zufallsvariable X_j:

$$X_j = \begin{cases} 1 & \text{falls } h(k_j) = i, \\ 0 & \text{falls } h(k_j) \neq i. \end{cases}$$

Es gilt $\mathbf{Pr}[X_j = 1] = \mathbf{Pr}[h(k_j) = i] = 1/m$ und demnach auch $\mathbf{E}[X_j] = 1/m$.

4.3.2 Hashing mit verketteten Listen

Beweis: Suche nach k mit $i = h(k)$. Uns interessiert $E[|T[i]|]$.

Für jedes $j \in \{1, \ldots, n\}$ definiere Zufallsvariable X_j:

$$X_j = \begin{cases} 1 & \text{falls } h(k_j) = i, \\ 0 & \text{falls } h(k_j) \neq i. \end{cases}$$

Es gilt $\mathbf{Pr}[X_j = 1] = \mathbf{Pr}[h(k_j) = i] = 1/m$ und demnach auch $\mathbf{E}[X_j] = 1/m$.

$$\Rightarrow \mathbf{E}[|T[i]|]$$

4.3.2 Hashing mit verketteten Listen

Beweis: Suche nach k mit $i = h(k)$. Uns interessiert $E[|T[i]|]$.

Für jedes $j \in \{1, \ldots, n\}$ definiere Zufallsvariable X_j:

$$X_j = \begin{cases} 1 & \text{falls } h(k_j) = i, \\ 0 & \text{falls } h(k_j) \neq i. \end{cases}$$

Es gilt $\mathbf{Pr}[X_j = 1] = \mathbf{Pr}[h(k_j) = i] = 1/m$ und demnach auch $\mathbf{E}[X_j] = 1/m$.

$$\Rightarrow \mathbf{E}[|T[i]|] = \mathbf{E}\left[\sum_{j=1}^{n} X_j\right]$$

4.3.2 Hashing mit verketteten Listen

Beweis: Suche nach k mit $i = h(k)$. Uns interessiert $E[|T[i]|]$.

Für jedes $j \in \{1, \ldots, n\}$ definiere Zufallsvariable X_j:

$$X_j = \begin{cases} 1 & \text{falls } h(k_j) = i, \\ 0 & \text{falls } h(k_j) \neq i. \end{cases}$$

Es gilt $\mathbf{Pr}[X_j = 1] = \mathbf{Pr}[h(k_j) = i] = 1/m$ und demnach auch $\mathbf{E}[X_j] = 1/m$.

$$\Rightarrow \mathbf{E}[|T[i]|] = \mathbf{E}\left[\sum_{j=1}^{n} X_j\right] = \sum_{j=1}^{n} \mathbf{E}[X_j]$$

4.3.2 Hashing mit verketteten Listen

Beweis: Suche nach k mit $i = h(k)$. Uns interessiert $E[|T[i]|]$.

Für jedes $j \in \{1, \ldots, n\}$ definiere Zufallsvariable X_j:

$$X_j = \begin{cases} 1 & \text{falls } h(k_j) = i, \\ 0 & \text{falls } h(k_j) \neq i. \end{cases}$$

Es gilt $\mathbf{Pr}[X_j = 1] = \mathbf{Pr}[h(k_j) = i] = 1/m$ und demnach auch $\mathbf{E}[X_j] = 1/m$.

$$\Rightarrow \mathbf{E}[|T[i]|] = \mathbf{E}\left[\sum_{j=1}^{n} X_j\right] = \sum_{j=1}^{n} \mathbf{E}[X_j] = \frac{n}{m} = \alpha$$

Beweis: Suche nach k mit $i = h(k)$. Uns interessiert $E[|T[i]|]$.

Für jedes $j \in \{1, \dots, n\}$ definiere Zufallsvariable X_j:

$$X_j = \begin{cases} 1 & \text{falls } h(k_j) = i, \\ 0 & \text{falls } h(k_j) \neq i. \end{cases}$$

Es gilt $\mathbf{Pr}[X_j = 1] = \mathbf{Pr}[h(k_j) = i] = 1/m$ und demnach auch $\mathbf{E}[X_j] = 1/m$.

$$\Rightarrow \mathbf{E}[|T[i]|] = \mathbf{E}\left[\sum_{j=1}^{n} X_j\right] = \sum_{j=1}^{n} \mathbf{E}[X_j] = \frac{n}{m} = \alpha$$

Somit folgen wir bei der Suche nach dem Schlüssel k im Erwartungswert $1 + \alpha$ vielen Zeigern, wobei $+1$ für den Null-Zeiger des letzten Eintrages der Liste $T[i]$ steht. $\square$

4.3.2 Hashing mit verketteten Listen

Theorem 4.12

Unter der Annahme des uniformen Hashings benötigt eine erfolgreiche Suche nach einem uniform zufällig gewählten Schlüssel in der Hashtabelle im Erwartungswert eine Laufzeit von $\Theta(1 + \alpha)$.

4.3.2 Hashing mit verketteten Listen

Theorem 4.12

Unter der Annahme des uniformen Hashings benötigt eine erfolgreiche Suche nach einem uniform zufällig gewählten Schlüssel in der Hashtabelle im Erwartungswert eine Laufzeit von $\Theta(1+\alpha)$.

Beweis: Schlüssel $k_1, \ldots, k_n$ wurden in dieser Reihenfolge eingefügt.

4.3.2 Hashing mit verketteten Listen

Theorem 4.12

Unter der Annahme des uniformen Hashings benötigt eine erfolgreiche Suche nach einem uniform zufällig gewählten Schlüssel in der Hashtabelle im Erwartungswert eine Laufzeit von $\Theta(1 + \alpha)$.

Beweis: Schlüssel $k_1, \ldots, k_n$ wurden in dieser Reihenfolge eingefügt.

Wie vielen Zeigern folgen wir bei Suche nach zufällig ausgewähltem Schlüssel k_i?

4.3.2 Hashing mit verketteten Listen

Theorem 4.12

Unter der Annahme des uniformen Hashings benötigt eine erfolgreiche Suche nach einem uniform zufällig gewählten Schlüssel in der Hashtabelle im Erwartungswert eine Laufzeit von $\Theta(1+\alpha)$.

Beweis: Schlüssel $k_1, \ldots, k_n$ wurden in dieser Reihenfolge eingefügt.

Wie vielen Zeigern folgen wir bei Suche nach zufällig ausgewähltem Schlüssel k_i?
Da neue Elemente an den Anfang der Liste eingefügt werden, ist dazu nur von Interesse, wie viele Schlüssel **nach k_i** in die Liste $T[h(k_i)]$ eingefügt werden.

4.3.2 Hashing mit verketteten Listen

Theorem 4.12

Unter der Annahme des uniformen Hashings benötigt eine erfolgreiche Suche nach einem uniform zufällig gewählten Schlüssel in der Hashtabelle im Erwartungswert eine Laufzeit von $\Theta(1 + \alpha)$.

Beweis: Schlüssel $k_1, \ldots, k_n$ wurden in dieser Reihenfolge eingefügt.

Wie vielen Zeigern folgen wir bei Suche nach zufällig ausgewähltem Schlüssel k_i?
Da neue Elemente an den Anfang der Liste eingefügt werden, ist dazu nur von Interesse, wie viele Schlüssel **nach $\mathbf{k_i}$** in die Liste $T[h(k_i)]$ eingefügt werden.

Für jedes $i \in \{1, \ldots, n\}$ und jedes $j \in \{i + 1, \ldots, n\}$ definieren wir eine Zufallsvariable

$$X_{ij} = \begin{cases} 1 & \text{falls } h(k_j) = h(k_i), \\ 0 & \text{falls } h(k_j) \neq h(k_i). \end{cases}$$

Theorem 4.12

Unter der Annahme des uniformen Hashings benötigt eine erfolgreiche Suche nach einem uniform zufällig gewählten Schlüssel in der Hashtabelle im Erwartungswert eine Laufzeit von $\Theta(1+\alpha)$.

Beweis: Schlüssel $k_1, \dots, k_n$ wurden in dieser Reihenfolge eingefügt.

Wie vielen Zeigern folgen wir bei Suche nach zufällig ausgewähltem Schlüssel k_i?
Da neue Elemente an den Anfang der Liste eingefügt werden, ist dazu nur von Interesse, wie viele Schlüssel **nach k_i** in die Liste $T[h(k_i)]$ eingefügt werden.

Für jedes $i \in \{1, \dots, n\}$ und jedes $j \in \{i+1, \dots, n\}$ definieren wir eine Zufallsvariable

$$X_{ij} = \begin{cases} 1 & \text{falls } h(k_j) = h(k_i), \\ 0 & \text{falls } h(k_j) \neq h(k_i). \end{cases}$$

Es gilt $\mathbf{Pr}[X_{ij} = 1] = \mathbf{Pr}[h(k_i) = h(k_j)] = 1/m$ und demnach auch $\mathbf{E}[X_{ij}] = 1/m$.

4.3.2 Hashing mit verketteten Listen

Für festes $i \in \{1, \ldots, n\}$ gilt

$$\mathbf{E}\left[1 + \sum_{j=i+1}^{n} X_{ij}\right]$$

4.3.2 Hashing mit verketteten Listen

Für festes $i \in \{1, \ldots, n\}$ gilt

$$\mathbf{E}\left[1 + \sum_{j=i+1}^{n} X_{ij}\right] = 1 + \sum_{j=i+1}^{n} \mathbf{E}\left[X_{ij}\right]$$

4.3.2 Hashing mit verketteten Listen

Für festes $i \in \{1, \ldots, n\}$ gilt

$$\mathbf{E}\left[1 + \sum_{j=i+1}^{n} X_{ij}\right] = 1 + \sum_{j=i+1}^{n} \mathbf{E}\left[X_{ij}\right] = 1 + \frac{n-i}{m}.$$

4.3.2 Hashing mit verketteten Listen

Für festes $i \in \{1, \dots, n\}$ gilt

$$\mathbf{E}\left[1 + \sum_{j=i+1}^{n} X_{ij}\right] = 1 + \sum_{j=i+1}^{n} \mathbf{E}\left[X_{ij}\right] = 1 + \frac{n-i}{m}.$$

Durchschnitt über alle i:

$$\frac{1}{n}\sum_{i=1}^{n}\left(1 + \frac{n-i}{m}\right)$$

Für festes $i \in \{1, \ldots, n\}$ gilt

$$\mathbf{E}\left[1 + \sum_{j=i+1}^{n} X_{ij}\right] = 1 + \sum_{j=i+1}^{n} \mathbf{E}\left[X_{ij}\right] = 1 + \frac{n-i}{m}.$$

Durchschnitt über alle i:

$$\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n}\left(1 + \frac{n-i}{m}\right) &= \frac{1}{n}\left(n + \sum_{i=1}^{n}\frac{n-i}{m}\right) = 1 + \frac{1}{n}\sum_{i=1}^{n}\frac{n-i}{m} = 1 + \frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
&= 1 + \frac{1}{nm}\sum_{i=1}^{n-1} i = 1 + \frac{1}{nm}\cdot\frac{n(n-1)}{2} = 1 + \frac{n-1}{2m} = 1 + \frac{\alpha}{2} - \frac{1}{2m} = \Theta(1+\alpha).
\end{aligned}$$

Damit ist das Theorem bewiesen. $\square$

4.3.3 Geschlossenes Hashing

geschlossenes Hashing oder **Hashing mit offener Adressierung**:

Speichere alle Daten in Feld T.

geschlossenes Hashing oder **Hashing mit offener Adressierung**:
Speichere alle Daten in Feld T.

Betrachte dazu Hashfunktionen der Form

$$h\colon U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}.$$

4.3.3 Geschlossenes Hashing

geschlossenes Hashing oder **Hashing mit offener Adressierung**:
Speichere alle Daten in Feld T.

Betrachte dazu Hashfunktionen der Form

$$h\colon U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}.$$

Idee: Teste beim Einfügen die Positionen $h(k, 0)$, $h(k, 1)$, $h(k, 2)$ usw., bis freie Position gefunden.

4.3.3 Geschlossenes Hashing

geschlossenes Hashing oder **Hashing mit offener Adressierung**:
Speichere alle Daten in Feld T.

Betrachte dazu Hashfunktionen der Form

$$h\colon U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}.$$

Idee: Teste beim Einfügen die Positionen $h(k, 0)$, $h(k, 1)$, $h(k, 2)$ usw., bis freie Position gefunden.

Annahme: $h(k, 0), h(k, 1), \ldots, h(k, m-1)$ ist Permutation von $\{0, 1, \ldots, m-1\}$.

4.3.3 Geschlossenes Hashing

```
INSERT(x)
for (int i = 0; i < m; i++) {
    j = h(x.key, i);
    if (T[j] == null) {
        T[j] = x;
        return j;
    }
}
return „Überlauf“;
```

4.3.3 Geschlossenes Hashing

```
INSERT(x)
for (int i = 0; i < m; i++) {
    j = h(x.key, i);
    if (T[j] == null) {
        T[j] = x;
        return j;
    }
}
return „Überlauf“;
```

0	1	2	3	4	5	6	7
	#		#		#	#	

4.3.3 Geschlossenes Hashing

```
INSERT(x)
for (int i = 0; i < m; i++) {
    j = h(x.key, i);
    if (T[j] == null) {
        T[j] = x;
        return j;
    }
}
return „Überlauf“;
```

0	1	2	3	4	5	6	7
	#		#		#	#	

$h(k, 0) = 1$

4.3.3 Geschlossenes Hashing

```
INSERT(x)
for (int i = 0; i < m; i++) {
    j = h(x.key, i);
    if (T[j] == null) {
        T[j] = x;
        return j;
    }
}
return „Überlauf“;
```

0	1	2	3	4	5	6	7
	#		#		#	#	

$h(k, 1) = 3$

4.3.3 Geschlossenes Hashing

```
INSERT(x)
for (int i = 0; i < m; i++) {
    j = h(x.key, i);
    if (T[j] == null) {
        T[j] = x;
        return j;
    }
}
return „Überlauf“;
```

0	1	2	3	4	5	6	7
	#		#		#	#	

$h(k, 2) = 0$

4.3.3 Geschlossenes Hashing

```
INSERT(x)
for (int i = 0; i < m; i++) {
    j = h(x.key, i);
    if (T[j] == null) {
        T[j] = x;
        return j;
    }
}
return „Überlauf";
```

0	1	2	3	4	5	6	7
k	#		#		#	#	

$h(k, 2) = 0$

4.3.3 Geschlossenes Hashing

```
INSERT(x)
for (int i = 0; i < m; i++) {
        j = h(x.key, i);
        if (T[j] == null) {
                T[j] = x;
                return j;
        }
}
return „Überlauf“;
```

```
SEARCH(k)
for (int i = 0; i < m; i++) {
        j = h(k, i);
        if ((T[j] ≠ null) && (T[j].key == k)) {
                return j;
        } else if (T[j] == null) {
                return −1; //„nicht vorhanden“;
        }
}
return −1; // „nicht vorhanden“;
```

0	1	2	3	4	5	6	7
k	#		#		#	#	

$h(k,2) = 0$

4.3.3 Geschlossenes Hashing

```
INSERT(x)
for (int i = 0; i < m; i++) {
    j = h(x.key, i);
    if (T[j] == null) {
        T[j] = x;
        return j;
    }
}
return „Überlauf“;
```

0	1	2	3	4	5	6	7
k	#		#		#	#	

$h(k, 2) = 0$

```
SEARCH(k)
for (int i = 0; i < m; i++) {
    j = h(k, i);
    if ((T[j] ≠ null) && (T[j].key == k)) {
        return j;
    } else if (T[j] == null) {
        return −1; //„nicht vorhanden“;
    }
}
return −1; // „nicht vorhanden“;
```

0	1	2	3	4	5	6	7
k	#		#		#	#	

4.3.3 Geschlossenes Hashing

```
INSERT(x)
for (int i = 0; i < m; i++) {
    j = h(x.key, i);
    if (T[j] == null) {
        T[j] = x;
        return j;
    }
}
return „Überlauf“;
```

```
SEARCH(k)
for (int i = 0; i < m; i++) {
    j = h(k, i);
    if ((T[j] ≠ null) && (T[j].key == k)) {
        return j;
    } else if (T[j] == null) {
        return −1; //„nicht vorhanden“;
    }
}
return −1; // „nicht vorhanden“;
```

0	1	2	3	4	5	6	7
k	#		#		#	#	

$h(k, 2) = 0$

0	1	2	3	4	5	6	7
k	#		#		#	#	

$h(k, 0) = 1$

4.3.3 Geschlossenes Hashing

INSERT(x)

```
for (int i = 0; i < m; i++) {
    j = h(x.key, i);
    if (T[j] == null) {
        T[j] = x;
        return j;
    }
}
return „Überlauf“;
```

SEARCH(k)

```
for (int i = 0; i < m; i++) {
    j = h(k, i);
    if ((T[j] ≠ null) && (T[j].key == k)) {
        return j;
    } else if (T[j] == null) {
        return −1; //„nicht vorhanden“;
    }
}
return −1; // „nicht vorhanden“;
```

0	1	2	3	4	5	6	7
k	#		#		#	#	

$h(k, 2) = 0$

0	1	2	3	4	5	6	7
k	#		#		#	#	

$h(k, 1) = 3$

4.3.3 Geschlossenes Hashing

```
INSERT(x)
for (int i = 0; i < m; i++) {
    j = h(x.key, i);
    if (T[j] == null) {
        T[j] = x;
        return j;
    }
}
return „Überlauf";
```

```
SEARCH(k)
for (int i = 0; i < m; i++) {
    j = h(k, i);
    if ((T[j] ≠ null) && (T[j].key == k)) {
        return j;
    } else if (T[j] == null) {
        return −1; //„nicht vorhanden";
    }
}
return −1; // „nicht vorhanden";
```

0	1	2	3	4	5	6	7
k	#		#		#	#	

$h(k, 2) = 0$

0	1	2	3	4	5	6	7
k	#		#		#	#	

$h(k, 2) = 0$

4.3.3 Geschlossenes Hashing

```
DELETE(k)
    j = SEARCH(k);
    if (j ≠ −1) {

        T[j] = null;
    }
```

4.3.3 Geschlossenes Hashing

```
DELETE(k)
    j = SEARCH(k);
    if (j ≠ −1) {

        T[j] = null;
    }
```

0	1	2	3	4	5	6	7
k	$\#$		k'		$\#$	$\#$	

4.3.3 Geschlossenes Hashing

```
DELETE(k)
    j = SEARCH(k);
    if (j ≠ −1) {

        T[j] = null;
    }
```

0	1	2	3	4	5	6	7
k	#		k'		#	#	

$\textsc{Delete}(k')$

4.3.3 Geschlossenes Hashing

```
DELETE(k)
    j = SEARCH(k);
    if (j ≠ −1) {

        T[j] = null;
    }
```

0	1	2	3	4	5	6	7
k	#				#	#	

$\textsc{Delete}(k')$

```
DELETE(k)
    j = SEARCH(k);
    if (j ≠ −1) {

        T[j] = null;
    }
```

0	1	2	3	4	5	6	7
k	#				#	#	

$h(k, 0) = 1$

4.3.3 Geschlossenes Hashing

```
DELETE(k)
    j = SEARCH(k);
    if (j ≠ −1) {

        T[j] = null;
    }
```

0	1	2	3	4	5	6	7
k	$\#$				$\#$	$\#$	

$h(k, 1) = 3$

4.3.3 Geschlossenes Hashing

```
DELETE(k)
    j = SEARCH(k);
    if (j ≠ −1) {

        T[j] = null;
    }
```

0	1	2	3	4	5	6	7
k	#				#	#	

$h(k, 2) = 0$

4.3.3 Geschlossenes Hashing

```
Delete(k)
    j = Search(k);
    if (j ≠ −1) {
        Del[j] = true;
        T[j] = null;
    }
```

0	1	2	3	4	5	6	7
k	#				#	#	

$h(k, 2) = 0$

4.3.3 Geschlossenes Hashing

```
DELETE(k)
    j = SEARCH(k);
    if (j ≠ −1) {
        Del[j] = true;
        T[j] = null;
    }
```

0	1	2	3	4	5	6	7
k	#				#	#	

$h(k, 2) = 0$

4.3.3 Geschlossenes Hashing

```
DELETE(k)
    j = SEARCH(k);
    if (j ≠ −1) {
        Del[j] = true;
        T[j] = null;
    }
```

```
SEARCH(k)
    for (int i = 0; i < m; i++) {
        j = h(k, i);
        if ((T[j] ≠ null) && (T[j].key == k)) {
            return j;
        } else if ((T[j] == null)                    ) {
            return −1; //„nicht vorhanden“;
        }
    }
    return −1; // „nicht vorhanden“;
```

0	1	2	3	4	5	6	7
k	#				#	#	

$h(k, 2) = 0$

4.3.3 Geschlossenes Hashing

```
DELETE(k)
    j = SEARCH(k);
    if (j ≠ −1) {
        Del[j] = true;
        T[j] = null;
    }
```

```
SEARCH(k)
    for (int i = 0; i < m; i++) {
        j = h(k, i);
        if ((T[j] ≠ null) && (T[j].key == k)) {
            return j;
        } else if ((T[j] == null) && (Del[j] == false) ) {
            return −1; //„nicht vorhanden“;
        }
    }
    return −1; // „nicht vorhanden“;
```

0	1	2	3	4	5	6	7
k	#				#	#	

$h(k,2) = 0$

4.3.3 Geschlossenes Hashing

Sondierungsreihenfolgen: gegeben sei „normale“ Hashfunktion $h' : U \to \{0, \ldots, m-1\}$

4.3.3 Geschlossenes Hashing

Sondierungsreihenfolgen: gegeben sei „normale“ Hashfunktion $h' : U \to \{0, \dots, m-1\}$

lineares Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k), h'(k)+1, h'(k)+2, \dots$, d. h.

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

Sondierungsreihenfolgen: gegeben sei „normale“ Hashfunktion $h' : U \to \{0, \ldots, m-1\}$

lineares Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k), h'(k)+1, h'(k)+2, \ldots$, d. h.

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

Nachteil: Tendenz, längere zusammenhängende Blöcke zu bilden:

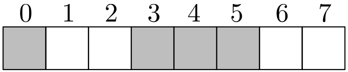
4.3.3 Geschlossenes Hashing

Sondierungsreihenfolgen: gegeben sei „normale“ Hashfunktion $h' : U \to \{0, \ldots, m-1\}$

lineares Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k), h'(k)+1, h'(k)+2, \ldots$, d. h.

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

Nachteil: Tendenz, längere zusammenhängende Blöcke zu bilden:



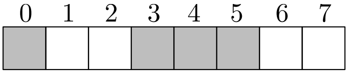
4.3.3 Geschlossenes Hashing

Sondierungsreihenfolgen: gegeben sei „normale“ Hashfunktion $h' : U \to \{0, \dots, m-1\}$

lineares Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k), h'(k)+1, h'(k)+2, \dots$, d. h.

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

Nachteil: Tendenz, längere zusammenhängende Blöcke zu bilden:



Einfügen eines neues Datums mit Schlüssel k an Position Z, wobei $h'(k)$ uniform zufällig

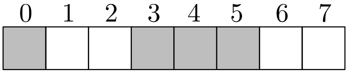
4.3.3 Geschlossenes Hashing

Sondierungsreihenfolgen: gegeben sei „normale“ Hashfunktion $h' : U \to \{0, \dots, m-1\}$

lineares Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k), h'(k)+1, h'(k)+2, \dots$, d. h.

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

Nachteil: Tendenz, längere zusammenhängende Blöcke zu bilden:



Einfügen eines neues Datums mit Schlüssel k an Position Z, wobei $h'(k)$ uniform zufällig

- $\mathbf{Pr}[Z=1] = \mathbf{Pr}[h'(k) \in \{0,1\}] = 2/8 = 1/4$

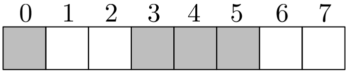
4.3.3 Geschlossenes Hashing

Sondierungsreihenfolgen: gegeben sei „normale“ Hashfunktion $h' : U \to \{0, \dots, m-1\}$

lineares Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k), h'(k) + 1, h'(k) + 2, \dots$, d. h.

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

Nachteil: Tendenz, längere zusammenhängende Blöcke zu bilden:



Einfügen eines neues Datums mit Schlüssel k an Position Z, wobei $h'(k)$ uniform zufällig

- $\mathbf{Pr}[Z = 1] = \mathbf{Pr}[h'(k) \in \{0, 1\}] = 2/8 = 1/4$
- $\mathbf{Pr}[Z = 2] = \mathbf{Pr}[h'(k) = 2] = 1/8$

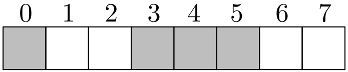
4.3.3 Geschlossenes Hashing

Sondierungsreihenfolgen: gegeben sei „normale“ Hashfunktion $h' : U \to \{0, \ldots, m-1\}$

lineares Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k)$, $h'(k)+1$, $h'(k)+2$, …, d. h.

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

Nachteil: Tendenz, längere zusammenhängende Blöcke zu bilden:



Einfügen eines neues Datums mit Schlüssel k an Position Z, wobei $h'(k)$ uniform zufällig

- $\mathbf{Pr}[Z = 1] = \mathbf{Pr}[h'(k) \in \{0,1\}] = 2/8 = 1/4$
- $\mathbf{Pr}[Z = 2] = \mathbf{Pr}[h'(k) = 2] = 1/8$
- $\mathbf{Pr}[Z = 6] = \mathbf{Pr}[h'(k) \in \{3,4,5,6\}] = 4/8 = 1/2$

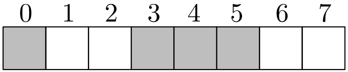
4.3.3 Geschlossenes Hashing

Sondierungsreihenfolgen: gegeben sei „normale“ Hashfunktion $h' : U \to \{0, \ldots, m-1\}$

lineares Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k), h'(k)+1, h'(k)+2, \ldots$, d. h.

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

Nachteil: Tendenz, längere zusammenhängende Blöcke zu bilden:



Einfügen eines neues Datums mit Schlüssel k an Position Z, wobei $h'(k)$ uniform zufällig

- $\mathbf{Pr}[Z = 1] = \mathbf{Pr}[h'(k) \in \{0,1\}] = 2/8 = 1/4$
- $\mathbf{Pr}[Z = 2] = \mathbf{Pr}[h'(k) = 2] = 1/8$
- $\mathbf{Pr}[Z = 6] = \mathbf{Pr}[h'(k) \in \{3,4,5,6\}] = 4/8 = 1/2$
- $\mathbf{Pr}[Z = 7] = \mathbf{Pr}[h'(k) = 7] = 1/8$

quadratisches Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k), h'(k) + 1^2, h'(k) + 2^2, h'(k) + 3^2, \ldots$ (jeweils modulo m).

4.3.3 Geschlossenes Hashing

quadratisches Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k), h'(k) + 1^2, h'(k) + 2^2, h'(k) + 3^2, \ldots$ (jeweils modulo m).

Gilt $h(k_1, 0) = h(k_2, 1)$, so werden (anders als beim linearen Sondieren) verschiedene Sequenzen durchlaufen. Es gilt insbesondere im Allgemeinen nicht $h(k_1, 1) = h(k_2, 2)$.

4.3.3 Geschlossenes Hashing

quadratisches Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k), h'(k)+1^2, h'(k)+2^2, h'(k)+3^2, \ldots$ (jeweils modulo m).

Gilt $h(k_1, 0) = h(k_2, 1)$, so werden (anders als beim linearen Sondieren) verschiedene Sequenzen durchlaufen. Es gilt insbesondere im Allgemeinen nicht $h(k_1, 1) = h(k_2, 2)$.

doppeltes Hashing: gegeben seien zwei „normale“ Hashfunktionen $h' : U \to \{0, \ldots, m-1\}$ und $h'' : U \to \{0, \ldots, m-1\}$

4.3.3 Geschlossenes Hashing

quadratisches Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k), h'(k) + 1^2, h'(k) + 2^2, h'(k) + 3^2, \ldots$ (jeweils modulo m).

Gilt $h(k_1, 0) = h(k_2, 1)$, so werden (anders als beim linearen Sondieren) verschiedene Sequenzen durchlaufen. Es gilt insbesondere im Allgemeinen nicht $h(k_1, 1) = h(k_2, 2)$.

doppeltes Hashing: gegeben seien zwei „normale“ Hashfunktionen $h' \colon U \to \{0, \ldots, m-1\}$ und $h'' \colon U \to \{0, \ldots, m-1\}$

Die Hashfunktion h ist dann definiert als

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

Analyse von geschlossenem Hashing: (ohne Löschen)

Analyse von geschlossenem Hashing: (ohne Löschen)

Uniformes Hashing: Beim Sondieren werden die Positionen in einer uniform zufälligen Reihenfolge getestet.

Analyse von geschlossenem Hashing: (ohne Löschen)

Uniformes Hashing: Beim Sondieren werden die Positionen in einer uniform zufälligen Reihenfolge getestet.

Sei $\alpha = n/m$ wieder der Auslastungsfaktor. Es gilt $\alpha \leq 1$.

Theorem 4.13

Unter der Annahme des uniformen Hashings untersucht eine erfolglose Suche nach einem Schlüssel, der sich nicht in der Hashtabelle befindet, beim geschlossenen Hashing im Erwartungswert höchstens $1/(1-\alpha)$ Positionen.

Beweis:

Es bezeichne X die Zufallsvariable, die die Anzahl untersuchter Positionen angibt.

4.3.3 Geschlossenes Hashing

Beweis:

Es bezeichne X die Zufallsvariable, die die Anzahl untersuchter Positionen angibt.

- Es gilt $\mathbf{Pr}[X \geq 1] = 1 = \alpha^0$, da in jedem Fall eine Position betrachtet wird.

4.3.3 Geschlossenes Hashing

Beweis:

Es bezeichne X die Zufallsvariable, die die Anzahl untersuchter Positionen angibt.

- Es gilt $\mathbf{Pr}[X \geq 1] = 1 = \alpha^0$, da in jedem Fall eine Position betrachtet wird.
- Es gilt $\mathbf{Pr}[X \geq 2] = \frac{n}{m} = \alpha^1$.

4.3.3 Geschlossenes Hashing

Beweis:

Es bezeichne X die Zufallsvariable, die die Anzahl untersuchter Positionen angibt.

- Es gilt $\mathbf{Pr}[X \geq 1] = 1 = \alpha^0$, da in jedem Fall eine Position betrachtet wird.
- Es gilt $\mathbf{Pr}[X \geq 2] = \frac{n}{m} = \alpha^1$.
- Es gilt $\mathbf{Pr}[X \geq 3] = \frac{n}{m} \cdot \frac{n-1}{m-1} \leq \alpha^2$, wobei $\frac{n-1}{m-1} \leq \frac{n}{m}$ aus $m \geq n$ folgt.

Beweis:

Es bezeichne X die Zufallsvariable, die die Anzahl untersuchter Positionen angibt.

- Es gilt $\mathbf{Pr}[X \geq 1] = 1 = \alpha^0$, da in jedem Fall eine Position betrachtet wird.
- Es gilt $\mathbf{Pr}[X \geq 2] = \frac{n}{m} = \alpha^1$.
- Es gilt $\mathbf{Pr}[X \geq 3] = \frac{n}{m} \cdot \frac{n-1}{m-1} \leq \alpha^2$, wobei $\frac{n-1}{m-1} \leq \frac{n}{m}$ aus $m \geq n$ folgt.
- Analog kann man für jedes $i \in \mathbb{N}$ argumentieren:

$$\mathbf{Pr}[X \geq i] = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+2}{m-i+2} \leq \alpha^{i-1}.$$

Beweis:

Es bezeichne X die Zufallsvariable, die die Anzahl untersuchter Positionen angibt.

- Es gilt $\mathbf{Pr}[X \geq 1] = 1 = \alpha^0$, da in jedem Fall eine Position betrachtet wird.
- Es gilt $\mathbf{Pr}[X \geq 2] = \frac{n}{m} = \alpha^1$.
- Es gilt $\mathbf{Pr}[X \geq 3] = \frac{n}{m} \cdot \frac{n-1}{m-1} \leq \alpha^2$, wobei $\frac{n-1}{m-1} \leq \frac{n}{m}$ aus $m \geq n$ folgt.
- Analog kann man für jedes $i \in \mathbb{N}$ argumentieren:

$$\mathbf{Pr}[X \geq i] = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+2}{m-i+2} \leq \alpha^{i-1}.$$

Aus der obigen Überlegung folgt

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} \mathbf{Pr}[X \geq i]$$

Beweis:

Es bezeichne X die Zufallsvariable, die die Anzahl untersuchter Positionen angibt.

- Es gilt $\mathbf{Pr}[X \geq 1] = 1 = \alpha^0$, da in jedem Fall eine Position betrachtet wird.
- Es gilt $\mathbf{Pr}[X \geq 2] = \frac{n}{m} = \alpha^1$.
- Es gilt $\mathbf{Pr}[X \geq 3] = \frac{n}{m} \cdot \frac{n-1}{m-1} \leq \alpha^2$, wobei $\frac{n-1}{m-1} \leq \frac{n}{m}$ aus $m \geq n$ folgt.
- Analog kann man für jedes $i \in \mathbb{N}$ argumentieren:

$$\mathbf{Pr}[X \geq i] = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+2}{m-i+2} \leq \alpha^{i-1}.$$

Aus der obigen Überlegung folgt

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} \mathbf{Pr}[X \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1}$$

Beweis:

Es bezeichne X die Zufallsvariable, die die Anzahl untersuchter Positionen angibt.

- Es gilt $\mathbf{Pr}[X \geq 1] = 1 = \alpha^0$, da in jedem Fall eine Position betrachtet wird.
- Es gilt $\mathbf{Pr}[X \geq 2] = \frac{n}{m} = \alpha^1$.
- Es gilt $\mathbf{Pr}[X \geq 3] = \frac{n}{m} \cdot \frac{n-1}{m-1} \leq \alpha^2$, wobei $\frac{n-1}{m-1} \leq \frac{n}{m}$ aus $m \geq n$ folgt.
- Analog kann man für jedes $i \in \mathbb{N}$ argumentieren:

$$\mathbf{Pr}[X \geq i] = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+2}{m-i+2} \leq \alpha^{i-1}.$$

Aus der obigen Überlegung folgt

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} \mathbf{Pr}[X \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1} \leq \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha},$$

womit das Theorem bewiesen ist. $\square$

4.3.3 Geschlossenes Hashing

Theorem 4.14

Unter der Annahme des uniformen Hashings untersucht eine erfolgreiche Suche nach einem uniform zufällig gewählten Schlüssel in der Hashtabelle im Erwartungswert höchstens $\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$ Positionen.

Theorem 4.14

Unter der Annahme des uniformen Hashings untersucht eine erfolgreiche Suche nach einem uniform zufällig gewählten Schlüssel in der Hashtabelle im Erwartungswert höchstens $\frac{1}{\alpha} \ln \left(\frac{1}{1-\alpha}\right)$ Positionen.

Beweis:

Suche nach Schlüssel x erzeugt die gleiche Sequenz wie das Einfügen des Schlüssels.

Theorem 4.14

Unter der Annahme des uniformen Hashings untersucht eine erfolgreiche Suche nach einem uniform zufällig gewählten Schlüssel in der Hashtabelle im Erwartungswert höchstens $\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$ Positionen.

Beweis:

Suche nach Schlüssel x erzeugt die gleiche Sequenz wie das Einfügen des Schlüssels.

Beim Einfügen des i-ten Schlüssels beträgt der Auslastungsfaktor $\alpha_i = (i-1)/m$

Theorem 4.14

Unter der Annahme des uniformen Hashings untersucht eine erfolgreiche Suche nach einem uniform zufällig gewählten Schlüssel in der Hashtabelle im Erwartungswert höchstens $\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$ Positionen.

Beweis:

Suche nach Schlüssel x erzeugt die gleiche Sequenz wie das Einfügen des Schlüssels.

Beim Einfügen des i-ten Schlüssels beträgt der Auslastungsfaktor $\alpha_i = (i-1)/m$ und demnach beträgt gemäß Theorem 4.13 die Anzahl der Positionen, die betrachtet werden, im Erwartungswert höchstens $1/(1-\alpha_i) = 1/(1-(i-1)/m)$.

Die Bildung des Durchschnitts über alle Schlüssel ergibt nun

$$
\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n}\frac{1}{1-(i-1)/m} = \frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{1-i/m} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\
&= \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \\
&\leq \frac{1}{\alpha}\int_{k=m-n}^{m}\frac{1}{x}\,dx \\
&= \frac{1}{\alpha}\ln\left(\frac{m}{m-n}\right) \\
&= \frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right),
\end{aligned}
$$

womit das Theorem bewiesen ist. $\square$

4.3.3 Geschlossenes Hashing