Algorithmen I
Tutorium 33

Woche 4 | 18. Mai 2018

Daniel Jungkind (daniel.jungkind@student.kit.edu)

INSTITUT FÜR THEORETISCHE INFORMATIK

Inhalt

Wahrscheinlichkeiten

Hashing

. . . mit verketteten Listen

. . . mit linearer Suche

WAS LETZTES MAL GESCHAH...

Quiz

Auf verketteten Listen läuft popBack in $O(1)$. **?**

Quiz

Auf verketteten Listen läuft popBack in $O(1)$. **Falsch.**

Auf **einfach verketteten** Listen nicht!

Quiz

Auf verketteten Listen läuft popBack in $O(1)$. **Falsch.**

Auf **einfach verketteten** Listen nicht!

Auf Unbounded Arrays lässt sich pushBack und popBack auch nicht-amortisiert in $O(1)$ implementieren. **?**

Quiz

Auf verketteten Listen läuft popBack in $O(1)$. **Falsch.**

Auf **einfach verketteten** Listen nicht!

Auf Unbounded Arrays lässt sich pushBack und popBack auch nicht-amortisiert in $O(1)$ implementieren. **Wahr.**

Man hält immer ein nächst-kleineres und ein nächst-größeres Array parat und kopiert das Schritt für Schritt bei jeder Operation mit. $\Rightarrow$ mehr Speicher

Quiz

Auf verketteten Listen läuft popBack in $O(1)$. **Falsch.**

Auf **einfach verketteten** Listen nicht!

Auf Unbounded Arrays lässt sich pushBack und popBack auch nicht-amortisiert in $O(1)$ implementieren. **Wahr.**

Man hält immer ein nächst-kleineres und ein nächst-größeres Array parat und kopiert das Schritt für Schritt bei jeder Operation mit. $\Rightarrow$ mehr Speicher

Eine amortisierte Laufzeitangabe liefert uns keine Laufzeitgarantien. **?**

Quiz

Auf verketteten Listen läuft popBack in $O(1)$. **Falsch.**

Auf **einfach verketteten** Listen nicht!

Auf Unbounded Arrays lässt sich pushBack und popBack auch nicht-amortisiert in $O(1)$ implementieren. **Wahr.**

Man hält immer ein nächst-kleineres und ein nächst-größeres Array parat und kopiert das Schritt für Schritt bei jeder Operation mit. $\Rightarrow$ mehr Speicher

Eine amortisierte Laufzeitangabe liefert uns keine Laufzeitgarantien. **Falsch.**

Doch: n Operationen laufen garantiert in der für n Operationen angegebenen Laufzeit ab. (Eine einzelne Op: keine Ahnung)

WAHRSCHEINLICHKEITEN

Probably useful

My little Wahrscheinlichkeitstheorie

- (Elementarer) **Grundraum** Ω = Menge aller möglichen Ergebnisse
- $A \subseteq \Omega$ heißt **Ereignis**, z. B.
  - *Elementarereignis* $\{\omega\}, \quad \omega \in \Omega$
  - *Sicheres* Ereignis Ω,
    unmögliches Ereignis $\varnothing$
  - Sei ω das Ergebnis eines konkreten Versuchs
    $\Longrightarrow$ A „tritt ein" $\Leftrightarrow \omega \in A$
- p_ω ist die **Wahrscheinlichkeit** von $\omega \in \Omega$,
  es gilt $\sum\limits_{\omega \in \Omega} p_\omega = 1$.
- **Gleichverteilung**: $p_\omega = \frac{1}{|\Omega|} \quad \forall \omega \in \Omega$.

My little Wahrscheinlichkeitstheorie

- Eine **Zufallsvariable** (ZV) X ist eine Abbildung, die jedem $\omega \in \Omega$ einen („beliebigen“) Wert in $\mathbb{R}$ zuweist, also $X : \Omega \longrightarrow \mathbb{R}$
- Seien X, Y ZVen:
  $X + Y$ ist wieder ZV, $\quad \lambda \cdot X$ auch ($\lambda \in \mathbb{R}$)
- **Erwartungswert** $\mathbb{E}$ einer Zufallsvariablen X (der „durchschnittlich eintretende“ Wert von X): $\mathbb{E}[X] := \sum_{\omega \in \Omega} p_\omega X(\omega)$
- Es gilt:

$$\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y],$$
$$\mathbb{E}[\lambda X] = \lambda \mathbb{E}[X]$$

$\Longrightarrow$ Der Erwartungswert ist eine *lineare* Abbildung!

My little Wahrscheinlichkeitstheorie

Karlsruher Institut für Technologie

- Eine **Zufallsvariable** (ZV) X ist eine Abbildung, die jedem $\omega \in \Omega$ einen („beliebigen“) Wert in $\mathbb{R}$ zuweist, also $X : \Omega \longrightarrow \mathbb{R}$
- Seien X, Y ZVen:
  $X + Y$ ist wieder ZV, $\quad \lambda \cdot X$ auch ($\lambda \in \mathbb{R}$)
- **Erwartungswert** $\mathbb{E}$ einer Zufallsvariablen X (der „durchschnittlich eintretende“ Wert von X): $\mathbb{E}[X] := \sum_{\omega \in \Omega} p_\omega X(\omega)$
- Es gilt:

$$\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y],$$
$$\mathbb{E}[\lambda X] = \lambda \mathbb{E}[X]$$

$\Longrightarrow$ Der Erwartungswert ist eine *lineare* Abbildung!

My little Wahrscheinlichkeitstheorie

- Eine **Zufallsvariable** (ZV) X ist eine Abbildung, die jedem $\omega \in \Omega$ einen („beliebigen“) Wert in $\mathbb{R}$ zuweist, also $X : \Omega \longrightarrow \mathbb{R}$
- Seien X, Y ZVen:
  $X + Y$ ist wieder ZV, $\quad \lambda \cdot X$ auch ($\lambda \in \mathbb{R}$)
- **Erwartungswert** $\mathbb{E}$ einer Zufallsvariablen X (der „durchschnittlich eintretende“ Wert von X): $\mathbb{E}[X] := \sum_{\omega \in \Omega} p_\omega X(\omega)$
- Es gilt:

$$\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y],$$
$$\mathbb{E}[\lambda X] = \lambda \mathbb{E}[X]$$

$\Longrightarrow$ Der Erwartungswert ist eine *lineare* Abbildung!

My little Wahrscheinlichkeitstheorie

Karlsruher Institut für Technologie

Beispiel 1

- 6-seitiger, fairer **Würfel** wird 1x geworfen
  $\Rightarrow \Omega := \{1, 2, 3, 4, 5, 6\}$, $p_\omega = \frac{1}{6}$ (gleichverteilt)
- Def. ZV X: Augenzahl eines Wurfes
  $\Rightarrow X(\omega) := \omega$
- Erwartungswert $\mathbb{E}[X] = 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + 3 \cdot \frac{1}{6} + 4 \cdot \frac{1}{6} + 5 \cdot \frac{1}{6} + 6 \cdot \frac{1}{6} = 3.5$
- Wahrscheinlichkeit, dass 2 oder 3 gewürfelt wird?
  $\Rightarrow \mathbb{P}[X = 2 \vee X = 3] = p_2 + p_3 = \frac{1}{3}$

My little Wahrscheinlichkeitstheorie

Beispiel 1

- 6-seitiger, fairer **Würfel** wird 1x geworfen
  $\Rightarrow \Omega := \{1, 2, 3, 4, 5, 6\}$, $p_\omega = \frac{1}{6}$ (gleichverteilt)
- Def. ZV X: Augenzahl eines Wurfes
  $\Rightarrow X(\omega) := \omega$
- Erwartungswert $\mathbb{E}[X] = 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + 3 \cdot \frac{1}{6} + 4 \cdot \frac{1}{6} + 5 \cdot \frac{1}{6} + 6 \cdot \frac{1}{6} = 3.5$
- Wahrscheinlichkeit, dass 2 oder 3 gewürfelt wird?
  $\Rightarrow \mathbb{P}[X = 2 \lor X = 3] = p_2 + p_3 = \frac{1}{3}$

My little Wahrscheinlichkeitstheorie

Karlsruher Institut für Technologie

Beispiel 1

- 6-seitiger, fairer **Würfel** wird 1x geworfen
  $\Rightarrow \Omega := \{1, 2, 3, 4, 5, 6\}$, $p_\omega = \frac{1}{6}$ (gleichverteilt)
- Def. ZV X: Augenzahl eines Wurfes
  $\Rightarrow X(\omega) := \omega$
- Erwartungswert $\mathbb{E}[X] = 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + 3 \cdot \frac{1}{6} + 4 \cdot \frac{1}{6} + 5 \cdot \frac{1}{6} + 6 \cdot \frac{1}{6} = 3.5$
- Wahrscheinlichkeit, dass 2 oder 3 gewürfelt wird?
  $\Rightarrow \mathbb{P}[X = 2 \vee X = 3] = p_2 + p_3 = \frac{1}{3}$

My little Wahrscheinlichkeitstheorie

Beispiel 1

- 6-seitiger, fairer **Würfel** wird 1x geworfen
  $\Rightarrow \Omega := \{1, 2, 3, 4, 5, 6\}$, $p_\omega = \frac{1}{6}$ (gleichverteilt)
- Def. ZV X: Augenzahl eines Wurfes
  $\Rightarrow X(\omega) := \omega$
- Erwartungswert $\mathbb{E}[X] = 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + 3 \cdot \frac{1}{6} + 4 \cdot \frac{1}{6} + 5 \cdot \frac{1}{6} + 6 \cdot \frac{1}{6} = 3.5$
- Wahrscheinlichkeit, dass 2 oder 3 gewürfelt wird?
  $\Rightarrow \mathbb{P}[X = 2 \vee X = 3] = p_2 + p_3 = \frac{1}{3}$

My little Wahrscheinlichkeitstheorie

Beispiel 1

- 6-seitiger, fairer **Würfel** wird 1x geworfen
  $\Rightarrow \Omega := \{1, 2, 3, 4, 5, 6\}$, $p_\omega = \frac{1}{6}$ (gleichverteilt)
- Def. ZV X: Augenzahl eines Wurfes
  $\Rightarrow X(\omega) := \omega$
- Erwartungswert $\mathbb{E}[X] = 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + 3 \cdot \frac{1}{6} + 4 \cdot \frac{1}{6} + 5 \cdot \frac{1}{6} + 6 \cdot \frac{1}{6} = 3.5$
- Wahrscheinlichkeit, dass 2 oder 3 gewürfelt wird?
  $\Rightarrow \mathbb{P}[X = 2 \vee X = 3] = p_2 + p_3 = \frac{1}{3}$

My little Wahrscheinlichkeitstheorie

Beispiel 2

- **Urne** mit Kugeln: 4 grüne, 1 rote. Einmal ziehen.
  $\Rightarrow \ \Omega = \{z_g, z_r\}$ (Ziehung grün, Ziehung rot) und $p_{z_g} = \frac{4}{5}$ und $p_{z_r} = \frac{1}{5}$.
- **Spiel**: Bei grün gewinnt man 2 €, bei rot 7 €. ZV: $G(z_g) := 2, \quad G(z_r) := 7$.
- Zu erwartender Gewinn?
  $\Rightarrow \ \mathbb{E}[G] = \frac{4}{5} \cdot 2 + \frac{1}{5} \cdot 7 = \frac{15}{5} = 3$

My little Wahrscheinlichkeitstheorie

Beispiel 2

- **Urne** mit Kugeln: 4 grüne, 1 rote. Einmal ziehen.
  $\Rightarrow \ \Omega = \{z_g, z_r\}$ (Ziehung grün, Ziehung rot) und
  $p_{z_g} = \frac{4}{5}$ und $p_{z_r} = \frac{1}{5}$.
- **Spiel**: Bei grün gewinnt man 2 €, bei rot 7 €. ZV:
  $G(z_g) := 2, \quad G(z_r) := 7.$
- Zu erwartender Gewinn?
  $\Rightarrow \ \mathbb{E}[G] = \frac{4}{5} \cdot 2 + \frac{1}{5} \cdot 7 = \frac{15}{5} = 3$

Beispiel 2

- **Urne** mit Kugeln: 4 grüne, 1 rote. Einmal ziehen.
  $\Rightarrow \ \Omega = \{z_g, z_r\}$ (Ziehung grün, Ziehung rot) und
  $p_{z_g} = \frac{4}{5}$ und $p_{z_r} = \frac{1}{5}$.
- **Spiel**: Bei grün gewinnt man 2 €, bei rot 7 €. ZV:
  $G(z_g) := 2, \quad G(z_r) := 7.$
- Zu erwartender Gewinn?
  $\Rightarrow \ \mathbb{E}[G] = \frac{4}{5} \cdot 2 + \frac{1}{5} \cdot 7 = \frac{15}{5} = 3$

My little Wahrscheinlichkeitstheorie

Karlsruher Institut für Technologie

Beispiel 2

- **Urne** mit Kugeln: 4 grüne, 1 rote. Einmal ziehen.
  $\Rightarrow \Omega = \{z_g, z_r\}$ (Ziehung grün, Ziehung rot) und
  $p_{z_g} = \frac{4}{5}$ und $p_{z_r} = \frac{1}{5}$.
- **Spiel**: Bei grün gewinnt man 2 €, bei rot 7 €. ZV:
  $G(z_g) := 2, \quad G(z_r) := 7$.
- Zu erwartender Gewinn?
  $\Rightarrow \mathbb{E}[G] = \frac{4}{5} \cdot 2 + \frac{1}{5} \cdot 7 = \frac{15}{5} = 3$

My little Wahrscheinlichkeitstheorie

Beispiel 2

- **Urne** mit Kugeln: 4 grüne, 1 rote. Einmal ziehen.
  $\Rightarrow \Omega = \{z_g, z_r\}$ (Ziehung grün, Ziehung rot) und
  $p_{z_g} = \frac{4}{5}$ und $p_{z_r} = \frac{1}{5}$.
- **Spiel**: Bei grün gewinnt man 2 €, bei rot 7 €. ZV:
  $G(z_g) := 2, \quad G(z_r) := 7$.
- Zu erwartender Gewinn?
  $\Rightarrow \mathbb{E}[G] = \frac{4}{5} \cdot 2 + \frac{1}{5} \cdot 7 = \frac{15}{5} = 3$

Einschub – Erwartete Laufzeit

Karlsruher Institut für Technologie

Ein Algorithmus mit tatsächlicher Laufzeit $T(n)$ hat **erwartete Laufzeit** in $O(g(n))$ $:\Leftrightarrow$ der Erwartungswert der Laufzeit liegt in $O(g(n))$.

Erwartete Laufzeit $\neq$ amortisierte Laufzeit!

„Wir haben bei n Operationen *amortisierte* Laufzeit $O(\cdots)$":
$\Rightarrow$ Wir haben das **garantiert!** (Über einzelne Operationen: keine Aussage!)

– vs. –

„Wir haben bei n Operationen *erwartete* Laufzeit $O(\cdots)$":
$\Rightarrow$ Wir haben das **wahrscheinlich**. Ist bloß ein Mittelwert. Kann auch drunter oder drüber liegen.

Einschub – Erwartete Laufzeit

Ein Algorithmus mit tatsächlicher Laufzeit $T(n)$ hat **erwartete Laufzeit** in $O(g(n)) :\Leftrightarrow$ der Erwartungswert der Laufzeit liegt in $O(g(n))$.

Erwartete Laufzeit $\neq$ amortisierte Laufzeit!

„Wir haben bei n Operationen *amortisierte* Laufzeit $O(\cdots)$“:
$\Rightarrow$ Wir haben das **garantiert**! (Über einzelne Operationen: keine Aussage!)

– vs. –

„Wir haben bei n Operationen *erwartete* Laufzeit $O(\cdots)$“:
$\Rightarrow$ Wir haben das **wahrscheinlich**. Ist bloß ein Mittelwert. Kann auch drunter oder drüber liegen.

HASHING

Das Genie beherrscht das Chaos

Hashtables: Übersicht

- **ungeordnete** (!) Datenstruktur
  $\Rightarrow$ Kein Index mehr (z.B. $A[i]$)   (Mit beliebigen Indizes erweiterbar!)
- **Operationen**: insert, remove, find
- Unordnung: **Wie?**
  $\Rightarrow$ Jedes Element e bekommt eindeutigen key(e) zugeordnet
  (Keys ab jetzt meistens *ganze Zahlen*)
  $\Rightarrow$ Benutzen Key, um das Element *schnell* zu wiederzufinden
- „Key **eindeutig**" heißt

$$\forall e, e' : \quad \mathrm{key}(e) \overset{<}{\underset{>}{=}} \mathrm{key}(e') \;\Leftrightarrow\; e \overset{<}{\underset{>}{=}} e'$$

Hashtables: Übersicht

- **ungeordnete** (!) Datenstruktur
  $\Rightarrow$ Kein Index mehr (z.B. $A[i]$)   (Mit beliebigen Indizes erweiterbar!)
- **Operationen**: insert, remove, find
- Unordnung: **Wie?**
  $\Rightarrow$ Jedes Element e bekommt eindeutigen key(e) zugeordnet
  (Keys ab jetzt meistens *ganze Zahlen*)
  $\Rightarrow$ Benutzen Key, um das Element *schnell* zu wiederzufinden
- „Key **eindeutig**“ heißt

$$\forall e, e' : \quad \text{key}(e) \overset{<}{\underset{>}{=}} \text{key}(e') \Leftrightarrow e \overset{<}{\underset{>}{=}} e'$$

Hashtables: Übersicht

- **ungeordnete** (!) Datenstruktur
  $\Rightarrow$ Kein Index mehr (z.B. $A[i]$)   (Mit beliebigen Indizes erweiterbar!)
- **Operationen**: insert, remove, find
- Unordnung: **Wie**?
  $\Rightarrow$ Jedes Element e bekommt eindeutigen $\text{key}(e)$ zugeordnet
  (Keys ab jetzt meistens *ganze Zahlen*)
  $\Rightarrow$ Benutzen Key, um das Element *schnell* zu wiederzufinden
- „Key **eindeutig**" heißt

$$\forall e, e' : \quad \text{key}(e) \overset{<}{\underset{>}{=}} \text{key}(e') \Leftrightarrow e \overset{<}{\underset{>}{=}} e'$$

Hashtables: Übersicht

- **ungeordnete** (!) Datenstruktur
  $\Rightarrow$ Kein Index mehr (z.B. $A[i]$) (Mit beliebigen Indizes erweiterbar!)
- **Operationen**: insert, remove, find
- Unordnung: **Wie**?
  $\Rightarrow$ Jedes Element e bekommt eindeutigen $\text{key}(e)$ zugeordnet
  (Keys ab jetzt meistens *ganze Zahlen*)
  $\Rightarrow$ Benutzen Key, um das Element *schnell* zu wiederzufinden
- „Key **eindeutig**" heißt

$$\forall e, e' : \quad \text{key}(e) \overset{<}{\underset{>}{=}} \text{key}(e') \Leftrightarrow e \overset{<}{\underset{>}{=}} e'$$

Hashtables: Under the hood

Aufbau

- Als Datenablage $\Rightarrow t : \textbf{array}[0 \ldots m-1]$ mit Länge m
- **Hashfunktion** $h : \mathbb{Z} \longrightarrow \{0, \ldots, m-1\}$
  Soll Elemente „gleichmäßig“ aufs Array verteilen

$\Rightarrow$ Element e landet in $t\left[h\left(\text{key}(e)\right)\right]$ (Invariante!)
(Wie genau: später)

- Perfektes, injektives h: Schön wär's!
  $\Rightarrow$ **Kollisionen** zu wahrscheinlich (zwei Elemente an selbe Stelle)

$\Rightarrow$ Rettung: z. B. **universelle** Hashfunktionen — Vorlesung:
Wenn $n \in O(m)$ Elemente in die Hashtable eingefügt werden $\Rightarrow$
erwartete $|\text{Kollisionen}| \in O(1)$

- Typische univ. Hashfunktion:
  $h_a(x) := (a \cdot x) \bmod m \quad (0 < a < m) \quad (m \text{ prim!})$

Hashtables: Under the hood

Aufbau

- Als Datenablage $\Rightarrow t :$ **array**$[0 \dots m-1]$ mit Länge m
- **Hashfunktion** $h : \mathbb{Z} \longrightarrow \{0, \dots, m-1\}$
  Soll Elemente „gleichmäßig“ aufs Array verteilen

$\Rightarrow$ Element e landet in $t\left[h(\text{key}(e))\right]$ (Invariante!)
(Wie genau: später)

– Perfektes, injektives h: Schön wär's!
$\Rightarrow$ **Kollisionen** zu wahrscheinlich (zwei Elemente an selbe Stelle)

$\Rightarrow$ Rettung: z. B. **universelle** Hashfunktionen — Vorlesung:
Wenn $n \in O(m)$ Elemente in die Hashtable eingefügt werden $\Rightarrow$ erwartete $|\text{Kollisionen}| \in O(1)$

- Typische univ. Hashfunktion:
  $h_a(x) := (a \cdot x) \textbf{ mod } m \quad (0 < a < m) \quad (m \text{ prim!})$

Hashtables: Under the hood

Aufbau

- Als Datenablage $\Rightarrow t$: **array**$[0 \ldots m-1]$ mit Länge m
- **Hashfunktion** $h : \mathbb{Z} \longrightarrow \{0, \ldots, m-1\}$
  Soll Elemente „gleichmäßig“ aufs Array verteilen

$\Rightarrow$ Element e landet in $t\left[h\left(\text{key}(e)\right)\right]$ (Invariante!)
(Wie genau: später)

$\Rightarrow$ Perfektes, injektives h: Schön wär's!
$\Rightarrow$ **Kollisionen** zu wahrscheinlich (zwei Elemente an selbe Stelle)

$\Rightarrow$ Rettung: z. B. **universelle** Hashfunktionen — Vorlesung:
Wenn $n \in O(m)$ Elemente in die Hashtable eingefügt werden $\Rightarrow$
erwartete $|\text{Kollisionen}| \in O(1)$

- Typische univ. Hashfunktion:
  $h_a(x) := (a \cdot x) \textbf{ mod } m \quad (0 < a < m) \quad (m \text{ prim!})$

Hashtables: Under the hood

Aufbau

- Als Datenablage $\Rightarrow$ t : **array**$[0 \ldots m-1]$ mit Länge m
- **Hashfunktion** $h : \mathbb{Z} \longrightarrow \{0, \ldots, m-1\}$
  Soll Elemente „gleichmäßig“ aufs Array verteilen

$\Rightarrow$ Element e landet in $t\left[h\left(\mathrm{key}(e)\right)\right]$ (Invariante!)
(Wie genau: später)

- Perfektes, injektives h: Schön wär's!
  $\Rightarrow$ **Kollisionen** zu wahrscheinlich (zwei Elemente an selbe Stelle)

$\Rightarrow$ Rettung: z. B. **universelle** Hashfunktionen — Vorlesung:
Wenn $n \in O(m)$ Elemente in die Hashtable eingefügt werden $\Rightarrow$
erwartete $|\text{Kollisionen}| \in O(1)$

- Typische univ. Hashfunktion:
  $h_a(x) := (a \cdot x) \textbf{ mod } m \quad (0 < a < m) \quad (m \text{ prim!})$

Hashtables: Under the hood

Aufbau

- Als Datenablage $\Rightarrow t :$ **array**$[0 \dots m-1]$ mit Länge m
- **Hashfunktion** $h : \mathbb{Z} \longrightarrow \{0, \dots, m-1\}$
  Soll Elemente „gleichmäßig“ aufs Array verteilen

$\Rightarrow$ Element e landet in $t\left[h\left(\text{key}(e)\right)\right]$ (Invariante!)
(Wie genau: später)

- Perfektes, injektives h: Schön wär’s!
  $\Rightarrow$ **Kollisionen** zu wahrscheinlich (zwei Elemente an selbe Stelle)

$\Rightarrow$ Rettung: z. B. **universelle** Hashfunktionen — Vorlesung:
Wenn $n \in O(m)$ Elemente in die Hashtable eingefügt werden $\Rightarrow$
erwartete $|\text{Kollisionen}| \in O(1)$

- Typische univ. Hashfunktion:
  $h_a(x) := (a \cdot x) \bmod m \quad (0 < a < m) \quad (m \text{ prim!})$

Hashtables: Under the hood

Aufbau

- Als Datenablage $\Rightarrow t$: **array**$[0 \dots m-1]$ mit Länge m
- **Hashfunktion** $h : \mathbb{Z} \longrightarrow \{0, \dots, m-1\}$
  Soll Elemente „gleichmäßig“ aufs Array verteilen

$\Rightarrow$ Element e landet in $t\left[h\left(\text{key}(e)\right)\right]$ (Invariante!)
(Wie genau: später)

- Perfektes, injektives h: Schön wär's!
  $\Rightarrow$ **Kollisionen** zu wahrscheinlich (zwei Elemente an selbe Stelle)

$\Rightarrow$ Rettung: z. B. **universelle** Hashfunktionen — Vorlesung:
Wenn $n \in O(m)$ Elemente in die Hashtable eingefügt werden $\Rightarrow$
erwartete $|\text{Kollisionen}| \in O(1)$

- Typische univ. Hashfunktion:
  $h_a(x) := (a \cdot x)\ \mathbf{mod}\ m \quad (0 < a < m) \quad (m \text{ prim!})$

Hashtables mit verketteten Listen

- Was tun bei Kollisionen?

$\Rightarrow$ Array-Slots von t bestehen aus **einfach verketteten Listen**

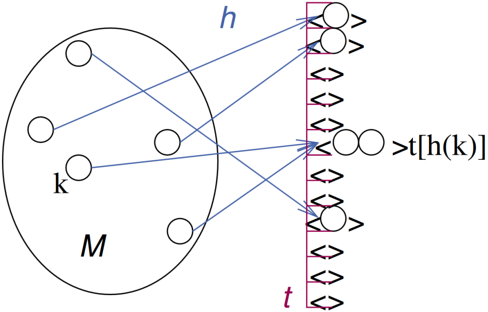
- **Mehrere** Items zu **einem** Slot zugeordnet?
  Alle rein in die Liste!

Hashtables mit verketteten Listen

- Was tun bei Kollisionen?

$\Rightarrow$ Array-Slots von t bestehen aus **einfach verketteten Listen**

- **Mehrere** Items zu **einem** Slot zugeordnet?
  Alle rein in die Liste!

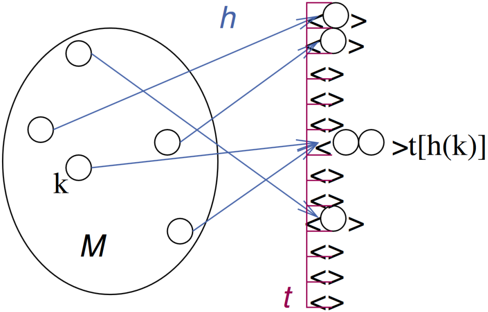


Hashtables mit verketteten Listen

- Was tun bei Kollisionen?

$\Rightarrow$ Array-Slots von *t* bestehen aus **einfach verketteten Listen**

- **Mehrere** Items zu **einem** Slot zugeordnet?
  Alle rein in die Liste!



Hashfunktionen allgemein

Bearbeitungshinweise

- Universelle Hashfunktion „herbeibeschwören“
  (z.B. auf Übungsblatt, in Klausur):
  „Sei h eine beliebige Hashfunktion aus der Familie universeller Hashfunktionen. Dann...“
- Ihr schreibt nen Algo. Es sollen k Elemente jeweils in erwartet $O(1)$ in eine Hashtable eingefügt werden.
  $\Rightarrow$ **Explizit** angeben, dass Größe der Hashtable in $\Omega(k)$ liegt!
  Sonst: Laufzeit **kaputt**, Satz aus VL greift dann **nicht**!

Hashtables mit verketteten Listen

Operation: insert(e : Element)

- Fügt das Element e an den **Anfang** der Liste beim Array-Index $h(\text{key}(e))$ ein
- Erwartete Laufzeit: $O(1)$
- Worst-Case: $O(1)$

Hashtables mit verketteten Listen

Operation: insert(e : Element)

- Fügt das Element e an den **Anfang** der Liste beim Array-Index $h(\text{key}(e))$ ein
- Erwartete Laufzeit: $O(1)$
- Worst-Case: $O(1)$

Hashtables mit verketteten Listen

Operation: remove($k : \mathbb{Z}$)

- Entfernt das Element e mit $\text{key}(e) = k$ aus der Liste beim Array-Index $h(k)$ (und gibt e zurück)
- Erwartete Laufzeit: $O(1)$
- Worst-Case: $O(n)$

Hashtables mit verketteten Listen

Operation: remove$(k : \mathbb{Z})$

- Entfernt das Element e mit $\text{key}(e) = k$ aus der Liste beim Array-Index $h(k)$ (und gibt e zurück)
- Erwartete Laufzeit: $O(1)$
- Worst-Case: $O(n)$

Hashtables mit verketteten Listen

Operation: find$(k : \mathbb{Z})$

- Sucht das Element e mit key$(e) = k$ aus der Liste beim Array-Index $h(k)$ und gibt e zurück
- Erwartete Laufzeit: $O(1)$
- Worst-Case: $O(n)$

Hashtables mit verketteten Listen

Operation: find($k : \mathbb{Z}$)

- Sucht das Element e mit key(e) $= k$ aus der Liste beim Array-Index $h(k)$ und gibt e zurück
- Erwartete Laufzeit: $O(1)$
- Worst-Case: $O(n)$

Hashtables mit verketteten Listen

Aufgabe 1: Verkettete Listen
Gegeben sei eine Hashtabelle der Größe m mit der Hashfunktion $h(x) := x \ \mathbf{mod} \ m$. Füge nacheinander folgende Elemente ein:
36, 78, 50, 1, 92, 15, 43, 99, 64 (hierbei gilt $\text{key}(e) := e$)
Dabei sei m zunächst 5 und danach 7.

Hashtables mit verketteten Listen

Lösung zu Aufgabe 1

$m = 5$:

0	1	2	3	4
$\langle 15, 50 \rangle$	$\langle 1, 36 \rangle$	$\langle 92 \rangle$	$\langle 43, 78 \rangle$	$\langle 64, 99 \rangle$

$m = 7$:

0	1	2	3	4	5	6
$\langle \rangle$	$\langle 64, 99, 43, 15, 92, 1, 50, 78, 36 \rangle$	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$

Achtung: Die Reihenfolge innerhalb der verketteten Liste entspricht jeweils der **umgekehrten** Einfügereihenfolge (da Elemente immer am **Anfang** der verketteten Liste hinzugefügt werden)

Hashtables mit linearer Suche

Karlsruher Institut für Technologie

- Verkettete Listen nicht *cachefreundlich* $\Rightarrow$ Geht es auch **ohne**?

$\Rightarrow$ Weg mit den Listen, stattdessen:

- **Einfügen**:
  Slot **besetzt**?
  $\Rightarrow$ lege Element beim **nächstbesten** freien Slot rechts davon ab.
  Rechts **alles besetzt**? Zwei Möglichkeiten:
  1. Array **zyklisch**: Beginne Suche von vorn
  2. Erweitere das Array um „**Pufferbereich**" mit Größe m'
     (auf den die Hashfunktion **nicht** abbildet)
- **Suche**: Starte Suche wo vermutet, weiterlaufen (und testen) bis e gefunden (falls $\perp$ erreicht: Abbruch)
- **Entfernen**: Suche das zu entfernende Element (wie oben), danach alle Elemente, die zu weit rechts liegen, nach links schieben
  ($\Rightarrow$ Leerstellen schließen!)

Hashtables mit linearer Suche

- Verkettete Listen nicht *cachefreundlich* $\Rightarrow$ Geht es auch **ohne**?

$\Rightarrow$ Weg mit den Listen, stattdessen:

- **Einfügen**:
  Slot **besetzt**?
  $\Rightarrow$ lege Element beim **nächstbesten** freien Slot rechts davon ab
  Rechts **alles besetzt**? Zwei Möglichkeiten:
  1. Array **zyklisch**: Beginne Suche von vorn
  2. Erweitere das Array um „**Pufferbereich**" mit Größe m'
     (auf den die Hashfunktion **nicht** abbildet)
- **Suche**: Starte Suche wo vermutet, weiterlaufen (und testen) bis e gefunden (falls $\perp$ erreicht: Abbruch)
- **Entfernen**: Suche das zu entfernende Element (wie oben), danach alle Elemente, die zu weit rechts liegen, nach links schieben ($\Rightarrow$ Leerstellen schließen!)

Hashtables mit linearer Suche

- Verkettete Listen nicht *cachefreundlich* $\Rightarrow$ Geht es auch **ohne**?

$\Rightarrow$ Weg mit den Listen, stattdessen:

- **Einfügen**:
  Slot **besetzt**?
  $\Rightarrow$ lege Element beim **nächstbesten** freien Slot rechts davon ab
  Rechts **alles besetzt**? Zwei Möglichkeiten:
  1. Array **zyklisch**: Beginne Suche von vorn
  2. Erweitere das Array um „**Pufferbereich**" mit Größe m'
     (auf den die Hashfunktion **nicht** abbildet)
- **Suche**: Starte Suche wo vermutet, weiterlaufen (und testen) bis e gefunden (falls $\perp$ erreicht: Abbruch)
- **Entfernen**: Suche das zu entfernende Element (wie oben), danach alle Elemente, die zu weit rechts liegen, nach links schieben
  ($\Rightarrow$ Leerstellen schließen!)

Hashtables mit linearer Suche

- Verkettete Listen nicht *cachefreundlich* $\Rightarrow$ Geht es auch **ohne**?

$\Rightarrow$ Weg mit den Listen, stattdessen:

- **Einfügen**:
  Slot **besetzt**?
  $\Rightarrow$ lege Element beim **nächstbesten** freien Slot rechts davon ab
  Rechts **alles besetzt**? Zwei Möglichkeiten:
  1. Array **zyklisch**: Beginne Suche von vorn
  2. Erweitere das Array um „**Pufferbereich**“ mit Größe m'
     (auf den die Hashfunktion **nicht** abbildet)
- **Suche**: Starte Suche wo vermutet, weiterlaufen (und testen) bis e gefunden (falls $\perp$ erreicht: Abbruch)
- **Entfernen**: Suche das zu entfernende Element (wie oben), danach alle Elemente, die zu weit rechts liegen, nach links schieben
  ($\Rightarrow$ Leerstellen schließen!)

Hashtables mit linearer Suche

Karlsruher Institut für Technologie

- Verkettete Listen nicht *cachefreundlich* $\Rightarrow$ Geht es auch **ohne**?

$\Rightarrow$ Weg mit den Listen, stattdessen:

- **Einfügen**:
  Slot **besetzt**?
  $\Rightarrow$ lege Element beim **nächstbesten** freien Slot rechts davon ab
  Rechts **alles besetzt**? Zwei Möglichkeiten:
  1. Array **zyklisch**: Beginne Suche von vorn
  2. Erweitere das Array um „**Pufferbereich**“ mit Größe m'
     (auf den die Hashfunktion **nicht** abbildet)
- **Suche**: Starte Suche wo vermutet, weiterlaufen (und testen) bis e gefunden (falls $\perp$ erreicht: Abbruch)
- **Entfernen**: Suche das zu entfernende Element (wie oben), danach alle Elemente, die zu weit rechts liegen, nach links schieben
  ($\Rightarrow$ Leerstellen schließen!)

Hashtables mit linearer Suche

- Verkettete Listen nicht *cachefreundlich* $\Rightarrow$ Geht es auch **ohne**?

$\Rightarrow$ Weg mit den Listen, stattdessen:

- **Einfügen**:
  Slot **besetzt**?
  $\Rightarrow$ lege Element beim **nächstbesten** freien Slot rechts davon ab
  Rechts **alles besetzt**? Zwei Möglichkeiten:
  1. Array **zyklisch**: Beginne Suche von vorn
  2. Erweitere das Array um „**Pufferbereich**“ mit Größe m'
     (auf den die Hashfunktion **nicht** abbildet)
- **Suche**: Starte Suche wo vermutet, weiterlaufen (und testen) bis e gefunden (falls $\perp$ erreicht: Abbruch)
- **Entfernen**: Suche das zu entfernende Element (wie oben), danach alle Elemente, die zu weit rechts liegen, nach links schieben
  ($\Rightarrow$ Leerstellen schließen!)

Hashtables mit linearer Suche

Aufgabe 2: Lineare Suche
Gegeben sei eine Hashtabelle der Größe $m = 8$ und Puffergröße $m' = 2$ mit der Hashfunktion $h(x) := x \mathbf{\ mod\ } m$. Füge nacheinander folgende Elemente ein:
36, 78, 50, 1, 92, 15, 43, 99, 64   (hierbei gilt $\text{key}(e) := e$)
Verwendet hierbei Hashing mit linearer Suche.
Was passiert, wenn im Anschluss die 43 wieder entfernt wird?

Hashtables mit linearer Suche

Lösung zu Aufgabe 2

	m								m'	
Index	0	1	2	3	4	5	6	7	—	—
Inhalt	64	1	50	43	36	92	78	15	99	

remove(43):
43 entfernt, stelle fest: 99 gehört eigentlich an Slot 3!
$\Rightarrow$ verschiebe 99 $\rightsquigarrow$ Slot 3. Also:

	m								m'	
Index	0	1	2	3	4	5	6	7	—	—
Inhalt	64	1	50	**99**	36	92	78	15		

Hashtables

Lineare Suche besser als Verkettete Listen?

+ Cachefreundlich
− Beschränkte Größe (Unbounded Array geht auch, aber mehr Sonderfälle)
− *insert* im Worst-Case in $\Theta(n)$
  Worst-Case wahrscheinlicher, weil potenziell mehr zu durchlaufen!

Hashtables

Lineare Suche besser als Verkettete Listen?

- + Cachefreundlich
- − Beschränkte Größe (Unbounded Array geht auch, aber mehr Sonderfälle)
- − *insert* im Worst-Case in $\Theta(n)$
  Worst-Case wahrscheinlicher, weil potenziell mehr zu durchlaufen!

Hashtables

Lineare Suche besser als Verkettete Listen?

- \+ Cachefreundlich
- − Beschränkte Größe (Unbounded Array geht auch, aber mehr Sonderfälle)
- − *insert* im Worst-Case in $\Theta(n)$
  Worst-Case wahrscheinlicher, weil potenziell mehr zu durchlaufen!

Hashtables

Lineare Suche besser als Verkettete Listen?

- \+ Cachefreundlich
- − Beschränkte Größe (Unbounded Array geht auch, aber mehr Sonderfälle)
- − *insert* im Worst-Case in $\Theta(n)$
  Worst-Case wahrscheinlicher, weil potenziell mehr zu durchlaufen!

Hashtables

Aufgabe 3a): Ein philosophischer Abend
Nach dem dritten Glas Vollmilch kommt ein Kommilitone auf die bahnbrechende Entdeckung, dass sich Hashing mit verketteten Listen hochgradig optimieren ließe, indem man die verketteten Listen stets sortiert halte.

Wie ändert sich das Worst-Case-Laufzeitverhalten von *insert*, *remove* und *find*?

Lösung zu Aufgabe 3a)

- *insert*: **Worst-Case** ändert sich zu $\Theta(n)$, wenn ein Element ganz am Ende eingefügt werden muss (und somit die ganze verkettete Liste durchlaufen wird)
- *remove* und *find*: **Binäre Suche** wäre ne Idee, aber bringt nichts: Index-Zugriff auf Listen nicht in $O(1) \Rightarrow$ binäre Suche nicht in $O(\log n)$
- Mit Skip-Lists (s. Übung) geht's auch schneller (zumindest amortisiert und je nachdem, wie balanciert die Datenstruktur gerade ist)

Hashtables

Lösung zu Aufgabe 3a)

- *insert*: **Worst-Case** ändert sich zu $\Theta(n)$, wenn ein Element ganz am Ende eingefügt werden muss (und somit die ganze verkettete Liste durchlaufen wird)
- *remove* und *find*: **Binäre Suche** wäre ne Idee, aber bringt nichts: Index-Zugriff auf Listen nicht in $O(1) \Rightarrow$ binäre Suche nicht in $O(\log n)$
- Mit Skip-Lists (s. Übung) geht's auch schneller (zumindest amortisiert und je nachdem, wie balanciert die Datenstruktur gerade ist)

Hashtables

Aufgabe 3b): Ein philosophischer Abend
Nachdem ihr einen gewissen Kommilitonen gefesselt und geknebelt auf dem Hinterhof zum Nachdenken abgelegt habt, diskutiert ihr, was passiert, wenn man die sortierten verketteten Listen durch sortierte unbeschränkte Arrays ersetzt.

Wie ändert sich nun das Worst-Case-Laufzeitverhalten von *insert*, *remove* und *find*?

Hashtables

Lösung zu Aufgabe 3b)

- *insert*: Im Worst-Case immer noch $\Theta(n)$. Zwar kann die Einfügestelle mit binärer Suche in $\Theta(\log n)$ gefunden werden, doch alle Elemente rechts davon müssen um eins verschoben werden
- *remove*: Auch weiterhin $\Theta(n)$. Binäre Suche hilft zwar beim Finden, aber alle Nachfolger müssen um eins nach links aufrücken
- *find*: Binäre Suche bringt „endlich“ etwas $\Rightarrow$ Laufzeit $\Theta(\log n)$

Hashtables

Lösung zu Aufgabe 3b)

- *insert*: Im Worst-Case immer noch $\Theta(n)$. Zwar kann die Einfügestelle mit binärer Suche in $\Theta(\log n)$ gefunden werden, doch alle Elemente rechts davon müssen um eins verschoben werden
- *remove*: Auch weiterhin $\Theta(n)$. Binäre Suche hilft zwar beim Finden, aber alle Nachfolger müssen um eins nach links aufrücken
- *find*: Binäre Suche bringt „endlich" etwas $\Rightarrow$ Laufzeit $\Theta(\log n)$

Hashtables

Karlsruher Institut für Technologie

Lösung zu Aufgabe 3b)

- *insert*: Im Worst-Case immer noch $\Theta(n)$. Zwar kann die Einfügestelle mit binärer Suche in $\Theta(\log n)$ gefunden werden, doch alle Elemente rechts davon müssen um eins verschoben werden
- *remove*: Auch weiterhin $\Theta(n)$. Binäre Suche hilft zwar beim Finden, aber alle Nachfolger müssen um eins nach links aufrücken
- *find*: Binäre Suche bringt „endlich" etwas $\Rightarrow$ Laufzeit $\Theta(\log n)$

Hashtables

Aufgabe 3c): Ein philosophischer Abend

Da außer euch zur späten Stunde keine anderen Kunden mehr vorhanden sind, gesellt sich der Milchbarkeeper zu euch und fragt, ob man da nicht noch was mit amortisierter Analyse machen kann.

Welches amortisierte Laufzeitverhalten lässt sich für *insert*, *remove* und *find* diagnostizieren (wenn man weiterhin sortierte unbeschränkte Arrays verwendet)?

Hashtables

Lösung zu Aufgabe 3c)

- *insert*: Betrachte eine Folge von n Einfügeoperationen, deren Elemente streng monoton kleiner werden (Worst-Case). Die k-te Einfügeoperationen muss $(k-1)$ Elemente verschieben
  $\Rightarrow$ n Operationen haben die Laufzeit $\Theta(n^2)$, da lässt sich nichts amortisieren
- *remove*: Zunächst ähnlich wie bei *insert*.
  Aber: Zu jedem *remove* in $O(n)$ gehört auch ein *insert* in $O(n)$. Also: Wälze den Aufwand von *remove* auf *insert* ab
  $\Rightarrow$ *remove* läuft amortisiert in $O(1)$. (Anmerkung: Einen praktischen Nutzen hat diese Betrachtungsweise leider nicht wirklich.)
- *find*: Hat immer die Laufzeit $\Theta(\log n)$, daran ändert auch Amortisierung nichts.

Hashtables

Lösung zu Aufgabe 3c)

- *insert*: Betrachte eine Folge von n Einfügeoperationen, deren Elemente streng monoton kleiner werden (Worst-Case). Die k-te Einfügeoperationen muss $(k-1)$ Elemente verschieben
$\Rightarrow$ n Operationen haben die Laufzeit $\Theta(n^2)$, da lässt sich nichts amortisieren
- *remove*: Zunächst ähnlich wie bei *insert*.
Aber: Zu jedem *remove* in $O(n)$ gehört auch ein *insert* in $O(n)$. Also: Wälze den Aufwand von *remove* auf *insert* ab
$\Rightarrow$ *remove* läuft amortisiert in $O(1)$. (Anmerkung: Einen praktischen Nutzen hat diese Betrachtungsweise leider nicht wirklich.)
- *find*: Hat immer die Laufzeit $\Theta(\log n)$, daran ändert auch Amortisierung nichts.

Hashtables

Lösung zu Aufgabe 3c)

- *insert*: Betrachte eine Folge von n Einfügeoperationen, deren Elemente streng monoton kleiner werden (Worst-Case). Die k-te Einfügeoperationen muss $(k-1)$ Elemente verschieben
$\Rightarrow$ n Operationen haben die Laufzeit $\Theta(n^2)$, da lässt sich nichts amortisieren
- *remove*: Zunächst ähnlich wie bei *insert*.
Aber: Zu jedem *remove* in $O(n)$ gehört auch ein *insert* in $O(n)$. Also: Wälze den Aufwand von *remove* auf *insert* ab
$\Rightarrow$ *remove* läuft amortisiert in $O(1)$. (Anmerkung: Einen praktischen Nutzen hat diese Betrachtungsweise leider nicht wirklich.)
- *find*: Hat immer die Laufzeit $\Theta(\log n)$, daran ändert auch Amortisierung nichts.

Komplexe Datenstrukturen

Aufgabe 4: SparseArray

Nehmt an, dass **allocate** euch beliebig viel *uninitialisierten* Speicher in konstanter Zeit liefert. Entwerft anhand dessen ein *SparseArray* mit folgenden Eigenschaften:

- Ein *SparseArray* mit n Slots braucht $O(n)$ Speicher
- Die Erzeugung eines leeren Arrays mit n Slots braucht $O(1)$ Zeit
- Eine Operation *reset*, die das *SparseArray* in $O(1)$ Zeit in leeren Zustand versetzt
- Die Operationen $get(i)$ und $set(i, x)$, die Zugriff auf den Inhalt am entspr. Index in $O(1)$ gewährleisten. Falls dieser leer ist, soll $\perp$ zurückgeliefert werden.

Komplexe Datenstrukturen

Lösung zu Aufgabe 4

Konstruiere ein *SparseArray* der Größe n mihtilfe von drei Arrays:
D (für die **Daten**), Z (für **Zähler**) und B (für **Beweise**).
Zudem initialisiere Counter $c := 0$.
$\Rightarrow$ **Speicheraufwand** in $O(n)$, **Erzeugung** läuft in $O(1)$ weil Speicherallokation in $O(1)$.
Operationen:

```
procedure set(i, x)
    D[i] := x
    Z[i] := c
    B[c] := i
    c++
```

(Durch Z und B entsteht somit ein rückversichernder Zirkelbezug, und c garantiert uns, bis zu welchem Index die Daten in B valide sind, d.h., $B[0 \ldots c-1]$ enthält alle gültigen Indizes).

Komplexe Datenstrukturen

Lösung zu Aufgabe 4

Konstruiere ein *SparseArray* der Größe n mihtilfe von drei Arrays: D (für die **Daten**), Z (für **Zähler**) und B (für **Beweise**).

Zudem initialisiere Counter $c := 0$.

$\Rightarrow$ **Speicheraufwand** in $O(n)$, **Erzeugung** läuft in $O(1)$ weil Speicherallokation in $O(1)$.

Operationen:

procedure set(i, x)
- $D[i] := x$
- $Z[i] := c$
- $B[c] := i$
- $c{+}{+}$

(Durch Z und B entsteht somit ein rückversichernder Zirkelbezug, und c garantiert uns, bis zu welchem Index die Daten in B valide sind, d.h., $B[0...c-1]$ enthält alle gültigen Indizes).

Komplexe Datenstrukturen

Lösung zu Aufgabe 4

procedure reset

    $c := 0$
    // Das markiert gemütlich den Inhalt aller Arrays als Unsinn

function get(i)

    if $Z[i] < c$ **and** $B[Z[i]] = i$ **then**
        return $D[i]$
    else
        // garbage inside $D[i]$, so:
        return $\perp$

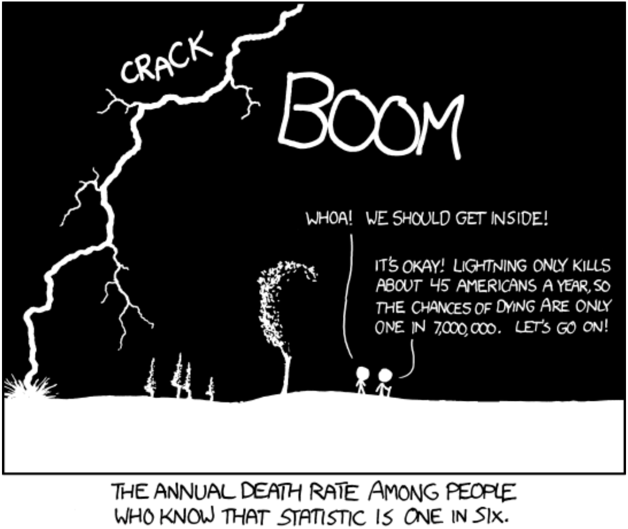
Komplexe Datenstrukturen

Lösung zu Aufgabe 4

```
procedure reset
    c := 0
    // Das markiert gemütlich den Inhalt aller Arrays als Unsinn

function get(i)
    if Z[i] < c and B[Z[i]] = i then
        return D[i]
    else
        // garbage inside D[i], so:
        return ⊥
```

Danke für die Aufmerksamkeit! ☺



http://xkcd.com/795