Karlsruher Institut für Technologie

3. Übung – Algorithmen I

INSTITUT FÜR THEORETISCHE INFORMATIK

Hashtabellen:

Beispielanwendung: Duplikaterkennung

Duplikaterkennung

Problem:

- Gegeben: Folge $A := \langle a_1, a_2, \ldots, a_n \rangle$ von Zahlen
- Frage: Enthält A ein oder mehrere Duplikate $a_i = a_j$, $i \neq j$?

Duplikaterkennung

Problem:

- Gegeben: Folge $A := \langle a_1, a_2, \ldots, a_n \rangle$ von Zahlen
- Frage: Enthält A ein oder mehrere Duplikate $a_i = a_j$, $i \neq j$?

Ansatz 1:

- Löse Problem mit Sortieren
- **Idee:**
  - Sortiere $A \rightsquigarrow A' := \langle a'_1, \ldots, a'_n \rangle$.
  - In A' stehen Duplikate nebeneinander
  - A' von links nach rechts durchlesen liefert alle Duplikate
  - **Worst-Case Laufzeit:** $\Omega(n \log n)$ (siehe Vorlesung „Sortieren & Co“)

Duplikaterkennung

Problem:

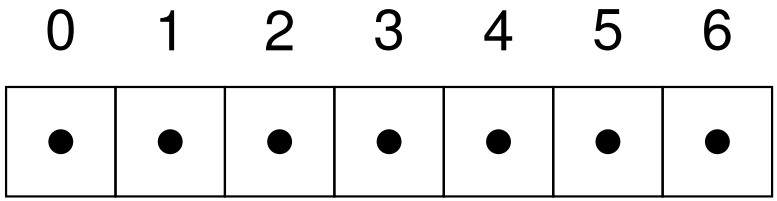
- Gegeben: Folge $A := \langle a_1, a_2, \ldots, a_n \rangle$ von Zahlen
- Frage: Enthält A ein oder mehrere Duplikate $a_i = a_j$, $i \neq j$?

Ansatz 2:

- Verwende eine Hashtabelle H (mit verketteten Listen)
- **Idee:**
  - Füge $a_1, \ldots, a_n$ nacheinander in H ein
  - Zahl schon drin: Duplikat erkannt!
  - **Laufzeit:** Erwartet $O(n)$
    - Bei zufälliger Hashfunktion **und**
    - wenn H mindestens $\Omega(n)$ Slots hat

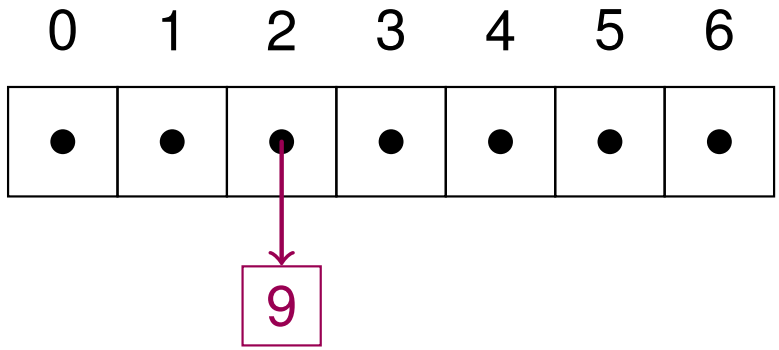
Erkennen von Duplikaten mittels Hashtabelle – Beispiel

Folge $A := \langle 9, 18, 42, 25, 33, 18, 104 \rangle$, Hashfunktion $h(a) = a \mod 7$



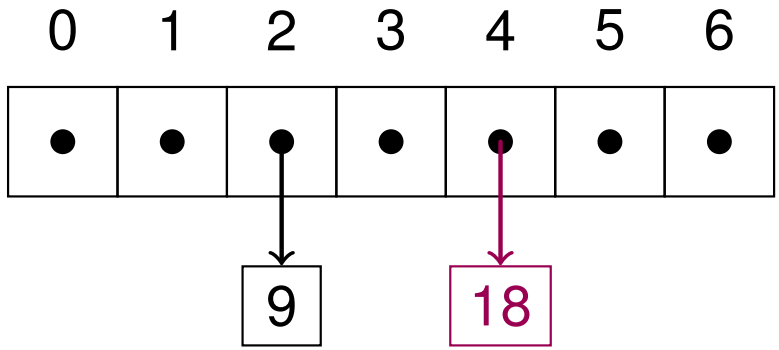
Erkennen von Duplikaten mittels Hashtabelle – Beispiel

Folge $A := \langle 9, 18, 42, 25, 33, 18, 104 \rangle$, Hashfunktion $h(a) = a \mod 7$



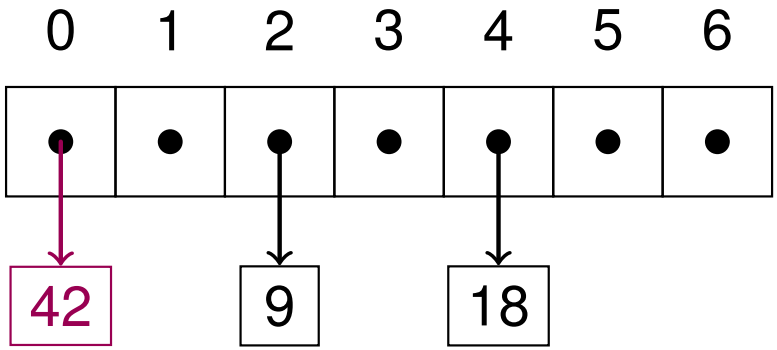
Erkennen von Duplikaten mittels Hashtabelle – Beispiel

Folge $A := \langle 9, 18, 42, 25, 33, 18, 104 \rangle$, Hashfunktion $h(a) = a \mod 7$



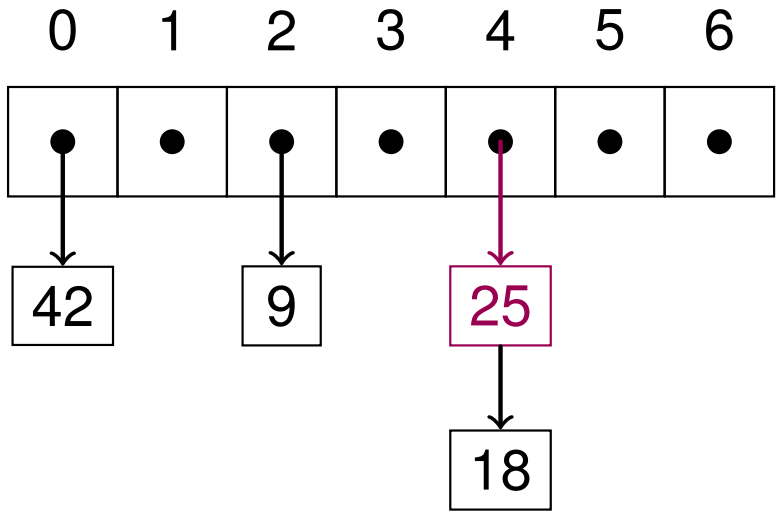
Erkennen von Duplikaten mittels Hashtabelle – Beispiel

Folge $A := \langle 9, 18, 42, 25, 33, 18, 104 \rangle$, Hashfunktion $h(a) = a \mod 7$



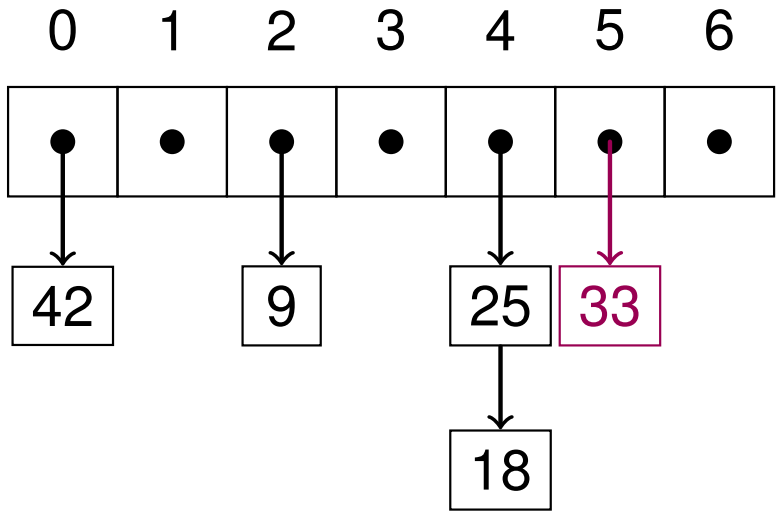
Erkennen von Duplikaten mittels Hashtabelle – Beispiel

Folge $A := \langle 9, 18, 42, 25, 33, 18, 104 \rangle$, Hashfunktion $h(a) = a \mod 7$



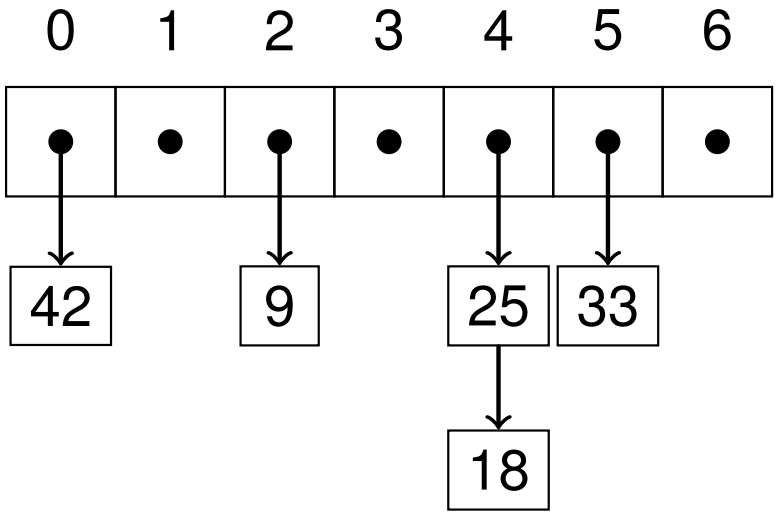
Erkennen von Duplikaten mittels Hashtabelle – Beispiel

Folge $A := \langle 9, 18, 42, 25, 33, 18, 104 \rangle$, Hashfunktion $h(a) = a \mod 7$



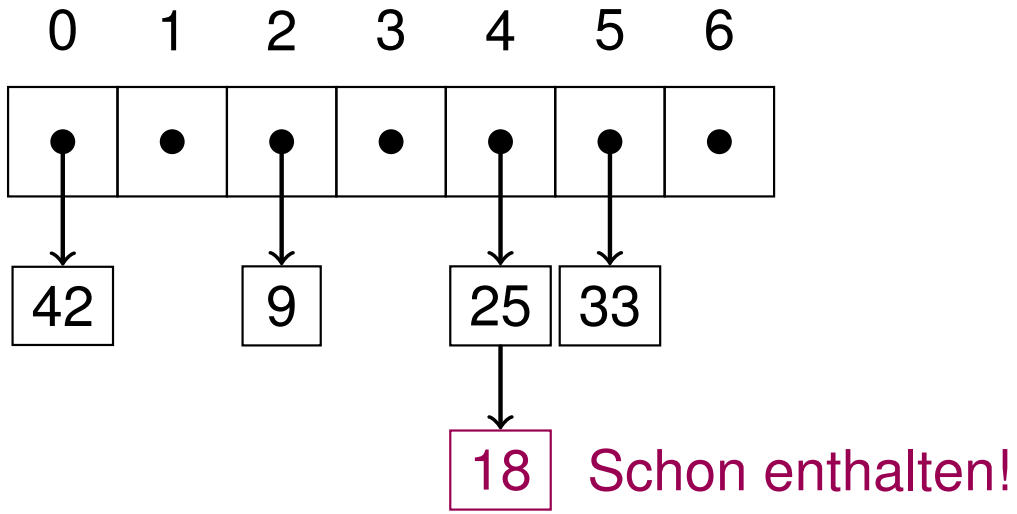
Erkennen von Duplikaten mittels Hashtabelle – Beispiel

Folge $A := \langle 9, 18, 42, 25, 33, 18, 104 \rangle$, Hashfunktion $h(a) = a \mod 7$



Erkennen von Duplikaten mittels Hashtabelle – Beispiel

Folge $A := \langle 9, 18, 42, 25, 33, 18, 104 \rangle$, Hashfunktion $h(a) = a \mod 7$



Erkennen von Duplikaten mittels Hashtabelle – Pseudocode

```
1: function hasDuplicates(A : Sequence of ℕ_0) : {true, false}
2:     H := new HashTableWithChaining of ℕ_0 with |A| slots
3:     for all a ∈ A do
4:         if H.find(a) ≠ NIL then return true
5:         H.insert(a)
6:     end do
7:     return false
```

Erkennen von Duplikaten mittels Hashtabelle – Laufzeitanalyse

Zwei Schritte

1. Erzeugen der Hashtabelle

2. Finden und Einfügen aller a_i aus $A = \langle a_1, \ldots, a_n \rangle$

Erkennen von Duplikaten mittels Hashtabelle – Laufzeitanalyse

Karlsruher Institut für Technologie

Zwei Schritte

1. Erzeugen der Hashtabelle
   - Alle Slots mit Null-Zeiger initialisieren
   - **Zeit:** $O(|A|)$, da Tabelle $|A|$ Slots hat
2. Finden und Einfügen aller a_i aus $A = \langle a_1, \ldots, a_n \rangle$

Erkennen von Duplikaten mittels Hashtabelle – Laufzeitanalyse

Karlsruher Institut für Technologie

Zwei Schritte

1. Erzeugen der Hashtabelle
   - Alle Slots mit Null-Zeiger initialisieren
   - **Zeit:** $O(|A|)$, da Tabelle $|A|$ Slots hat
2. Finden und Einfügen aller a_i aus $A = \langle a_1, \ldots, a_n \rangle$
   - **Zeit** pro Einfügen: $O(1)$
   - **Zeit** pro $H.find(a)$: $O(\text{Listenlänge})$
     - **Aber:** Erwartete Laufzeit $O(1)$
     - **Denn:** Tabelle hat $|A|$ Slots
     - **Hierzu** Satz aus Vorlesung: erwartete Listenlänge $O(1)$

$\Rightarrow$ **Gesamtzeit** für Finden und Einfügen: erwartet $O(|A|)$

Bloom Filter
Approximate Membership Tester

Bloom Filter

Randomisierte Datenstruktur

repräsentiert Menge M von n u-Bit Elementen
(mit Hilfe von $m \ll un$ Bits)

Operationen:

- *insert*(**Key** k)
- *contains*(**Key** k) : bool

Bloom Filter

Randomisierte Datenstruktur

repräsentiert Menge M von n u-Bit Elementen
(mit Hilfe von $m \ll un$ Bits)

Operationen:

- *insert*(**Key** k)
- *contains*(**Key** k) : bool

Ergebnis einer *contains*-Abfrage:

- *false*: $k \notin M$
- *true*: k ist wahrscheinlich in M, **false positive** mit Wahrscheinlichkeit f^+

Bloom Filter

Randomisierte Datenstruktur

repräsentiert Menge M von n u-Bit Elementen
(mit Hilfe von $m \ll un$ Bits)

Operationen:

- *insert*(**Key** k)
- *contains*(**Key** k) : bool

Ergebnis einer *contains*-Abfrage:

- *false*: $k \notin M$
- *true*: k ist wahrscheinlich in M, **false positive** mit Wahrscheinlichkeit f^+

Anwendungsbeispiele:

- Spellchecking
- Webcrawling
- Google Chrome Safe Browsing

Bloom Filter

Randomisierte Datenstruktur

- n Elemente
- $k \geq 1$ Hashfunktionen h_i
- array $A[0 \ldots, m-1]$ von Bits

Bloom Filter

Randomisierte Datenstruktur

- n Elemente
- $k \geq 1$ Hashfunktionen h_i
- array $A[0 \ldots, m-1]$ von Bits

insert(x):

1. setze $A[h_i(x)] = 1 \; \forall i \in \{1, \ldots, k\}$

Bloom Filter

Randomisierte Datenstruktur

- n Elemente
- $k \geq 1$ Hashfunktionen h_i
- array $A[0 \ldots, m-1]$ von Bits

insert(x):

1. setze $A[h_i(x)] = 1 \; \forall i \in \{1, \ldots, k\}$

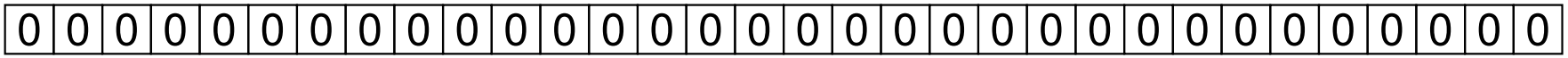
contains(x):

1. wenn $A[h_i(x)] = 1 \; \forall i \in \{1, \ldots, k\} \Rightarrow$ true
2. sonst $\Rightarrow$ false

Bloom Filter

Beispiel

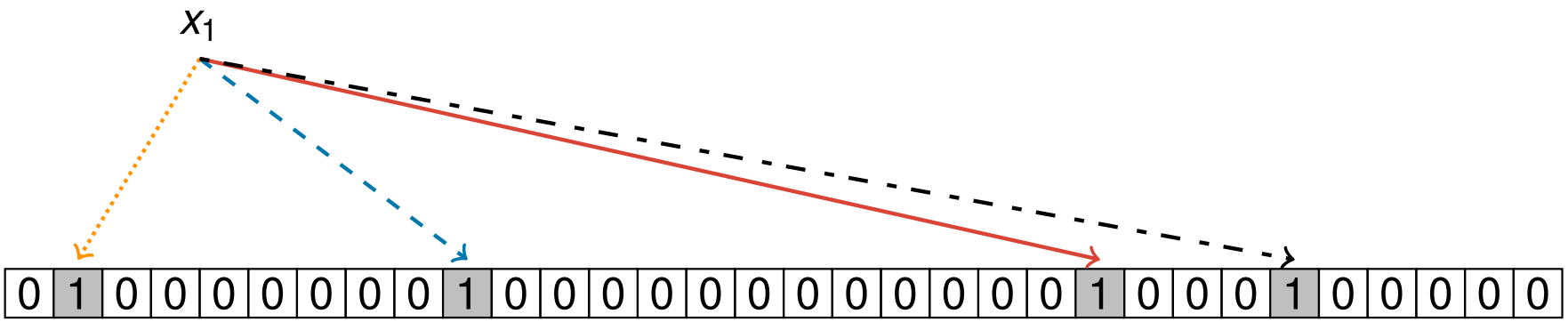
- $k = 4$ Hashfunktionen
- $m = 32$ Bits



Bloom Filter

Beispiel

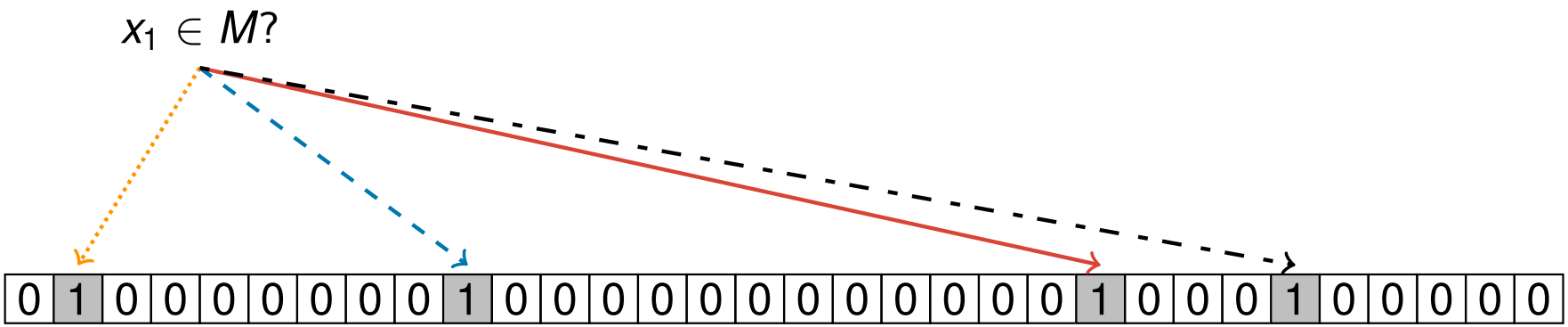
- *insert*(x_1):



Bloom Filter

Beispiel

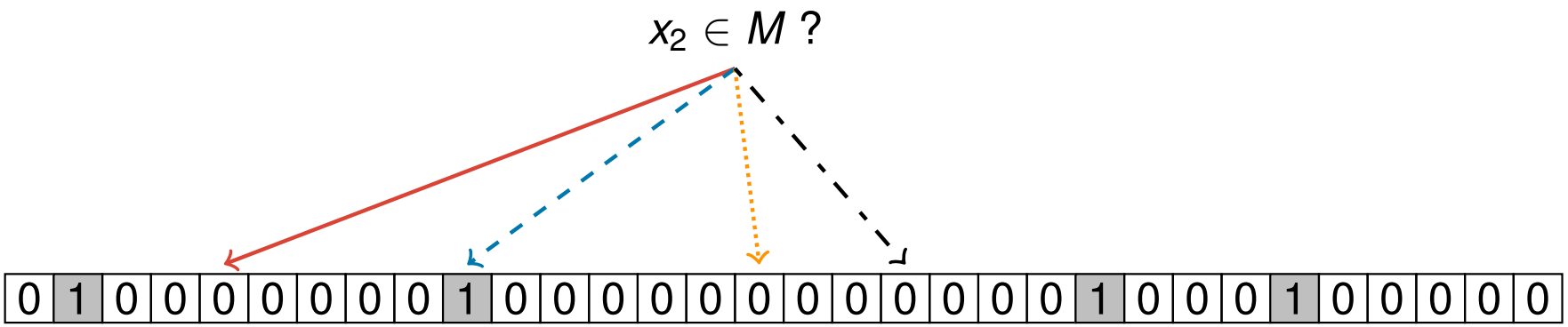
- contains(x_1) $\Rightarrow$ true



Bloom Filter

Beispiel

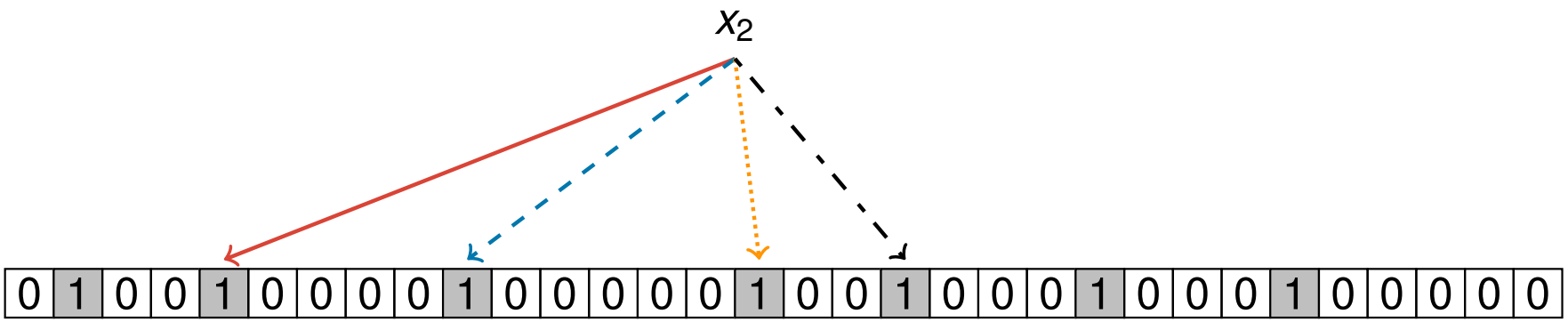
- contains(x_2) $\Rightarrow$ false



Bloom Filter

Beispiel

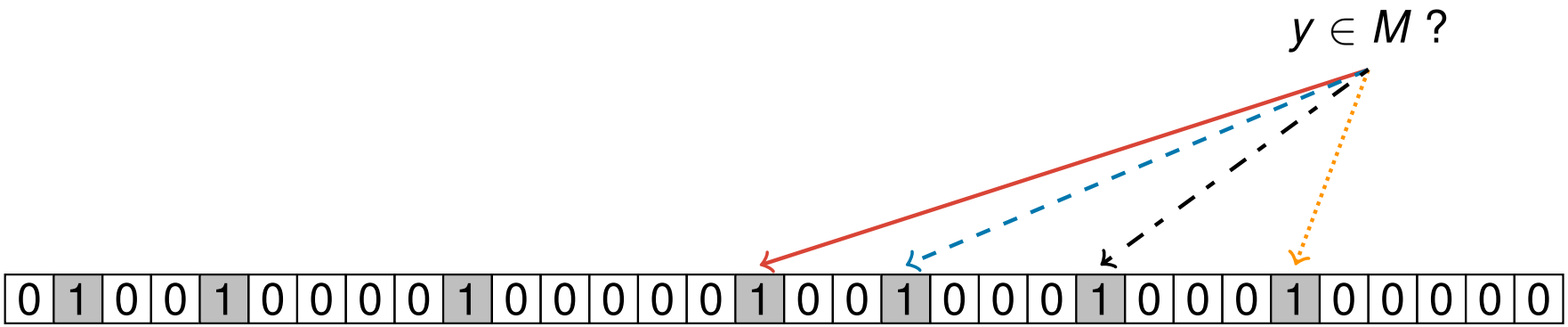
- insert(x_2)



Bloom Filter

Beispiel

- contains(y) $\Rightarrow$ true, obwohl $y \notin M$



Bloom Filter

Wie wahrscheinlich sind false positives?

- **Annahme:** Hashfunktionen haben uniform gleichverteiltes Bild
- p: WK, dass ein Bit = 1 nach n Einfügungen

$$p = 1 - \left(1 - \frac{1}{m}\right)^{kn}$$

Bloom Filter

Wie wahrscheinlich sind false positives?

- **Annahme:** Hashfunktionen haben uniform gleichverteiltes Bild
- p: WK, dass ein Bit = 1 nach n Einfügungen

$$p = 1 - \left(1 - \frac{1}{m}\right)^{kn}$$

Bloom Filter

Wie wahrscheinlich sind false positives?

- **Annahme:** Hashfunktionen haben uniform gleichverteiltes Bild
- p: WK, dass ein Bit = 1 nach n Einfügungen

$$p = 1 - \left(1 - \frac{1}{m}\right)^{kn}$$

Bloom Filter

Wie wahrscheinlich sind false positives?

- **Annahme:** Hashfunktionen haben uniform gleichverteiltes Bild
- p: WK, dass ein Bit = 1 nach n Einfügungen

$$p = 1 - \left(1 - \frac{1}{m}\right)^{kn}$$

Bloom Filter

Wie wahrscheinlich sind false positives?

- **Annahme:** Hashfunktionen haben uniform gleichverteiltes Bild
- p: WK, dass ein Bit = 1 nach n Einfügungen

$$p = 1 - \left(1 - \frac{1}{m}\right)^{kn}$$

Bloom Filter

Wie wahrscheinlich sind false positives?

- **Annahme:** Hashfunktionen haben uniform gleichverteiltes Bild
- p: WK, dass ein Bit = 1 nach n Einfügungen

$$p = 1 - \left(1 - \tfrac{1}{m}\right)^{kn} = 1 - \left(\left(1 - \tfrac{1}{m}\right)^{m}\right)^{kn/m}$$

Bloom Filter

Wie wahrscheinlich sind false positives?

- **Annahme:** Hashfunktionen haben uniform gleichverteiltes Bild
- p: WK, dass ein Bit = 1 nach n Einfügungen

$$p = 1 - \left(1 - \tfrac{1}{m}\right)^{kn} = 1 - \left(\left(1 - \tfrac{1}{m}\right)^{m}\right)^{kn/m} \approx 1 - e^{-kn/m}$$

Bloom Filter

Wie wahrscheinlich sind false positives?

- **Annahme:** Hashfunktionen haben uniform gleichverteiltes Bild
- p: WK, dass ein Bit = 1 nach n Einfügungen

$$p = 1 - \left(1 - \tfrac{1}{m}\right)^{kn} = 1 - \left(\left(1 - \tfrac{1}{m}\right)^{m}\right)^{kn/m} \approx 1 - e^{-kn/m}$$

- WK für false positive $\Leftrightarrow$ WK, dass alle $\boldsymbol{k}$ Hash-Bits 1 sind

$$\Rightarrow f^{+} \approx \left(1 - e^{-kn/m}\right)^{k}$$

Bloom Filter

Wie wahrscheinlich sind false positives?

- **Annahme:** Hashfunktionen haben uniform gleichverteiltes Bild
- p: WK, dass ein Bit = 1 nach n Einfügungen

$$p = 1 - \left(1 - \tfrac{1}{m}\right)^{kn} = 1 - \left(\left(1 - \tfrac{1}{m}\right)^{m}\right)^{kn/m} \approx 1 - e^{-kn/m}$$

- WK für false positive $\Leftrightarrow$ WK, dass alle $\boldsymbol{k}$ Hash-Bits 1 sind

$$\Rightarrow f^{+} \approx \left(1 - e^{-kn/m}\right)^{k}$$

- optimialer Wert für $k := \frac{m}{n} \ln 2$

Bloom Filter

Rechenbeispiel

Wahrscheinlichkeit für false positive: $\left(1 - e^{-kn/m}\right)^k$

- $n = 100\,000$ Objekte
- $m = 1\,000\,000$ Bits
- wähle $k = 7$
- Wahrscheinlichkeit für false positive < 1%

Vorteile:

- viel kompaktere Repräsentation als Abspeichern von Objekten
- z.B. Strings, Webseiten, Hashwerte, ...

Unbounded Hashtables

Unbounded Hashtables

Amortisierung

Problem

- Anzahl der einzufügenden Elemente nicht bekannt

Was passiert wenn eine Hashtabelle zu voll wird?

- Hashing mit linearer Suche: Überlauf
- Hashing mit verk. Liste: Verlangsamerung der Operationen

Lösung: Hashtabelle dynamisch vergrößern und verkleinern

Unbounded Hashtables

mit verketteten Listen

Modifizierte Operationen

- *find*: keine Veränderung
- *insert*: Größe verdoppeln, bei #Slots Elemente
- *remove*: Größe halbieren, bei $\frac{1}{4}$#Slots Elemente

Erinnert an unbeschränkte Arrays

Unbounded Hashtables

mit verketteten Listen

Problem:

- Hashfunktion muss zur Tabellengröße passen
- **Grund:** Soll möglichst gleichverteilt streuen
- Nach Größenänderung nicht mehr der Fall

Lösung: Bei Größenänderung neue Hashfunktion wählen

- **Dann:** vollständiger „rehash“
- **D.h.:** Elemente nicht nur kopieren, sondern alle neu einfügen

Unbounded Hashtable

Laufzeit

Laufzeit von *insert, find, remove* (exkl. rehash):

- Unverändert erwartet O(1)

Laufzeit von rehash:

- Amortisiert O(1)
- Argumentation wie bei unbeschränkten Arrays
- Bankkontomethode

Neue Hashfunktion wählen

Beispiel

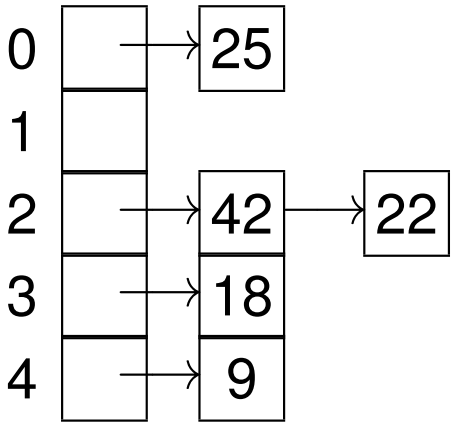
Hashen von Zahlen

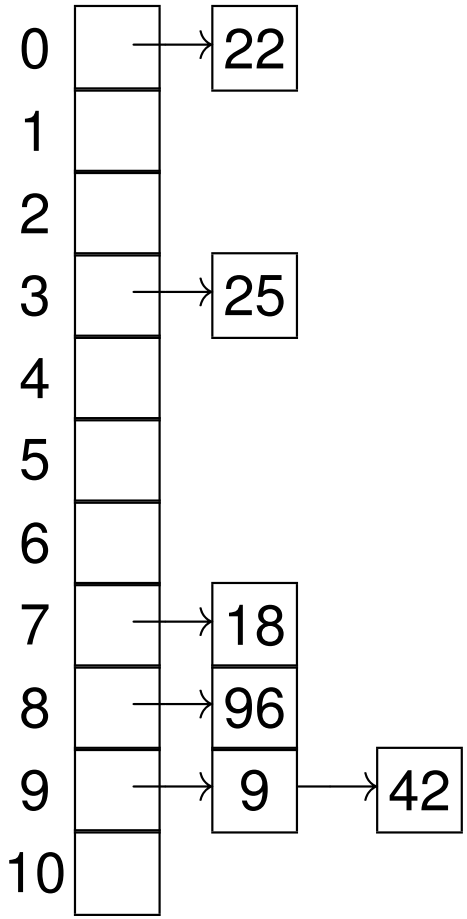
- $h(x) = x \mod$ Tabellengröße
  - Problem: Tabellengröße $= 2^k$
  - Entspricht Extrahieren der k niedrigsten Bits
  - Nur k niedrigsten Bits nehmen Einfluss
- **Besser:**
  - Tabellengröße immer Primzahl
  - Möglichst weit entfernt von Zweierpotenzen
- **Implementierung:**
  - Primzahlentabelle
  - Wähle bei Größenänderungen die nächstgrößere Primzahl aus Tabelle

Rehash

Beispiel

- insert: 22, 42, 9, 25, 18 und 96
- $h_1(x) = x \bmod 5$, $h_2(x) = x \bmod 11$





Hashing von Zeichenketten

Nicht: kryptographische Message Digests (MD5, SHA, etc)!

Hashing von Zeichenketten

Gegeben Zeichenkette $s = \langle x_0, x_1, \ldots, x_{n-1} \rangle$.
Ganz schlechte Hashfunktion:

$$h(s) = \sum_{i=0}^{n-1} x_i \mod 2^k$$

Hashing von Zeichenketten

Gegeben Zeichenkette $s = \langle x_0, x_1, \ldots, x_{n-1} \rangle$.
Ganz schlechte Hashfunktion:

$$h(s) = \sum_{i=0}^{n-1} x_i \mod 2^k$$

Etwas weniger schlechte Hashfunktion:

$$h(s) = 1 \cdot x_0 + 3 \cdot x_1 + 5 \cdot x_2 + 7 \cdot x_3 + \cdots \mod 2^k$$

Hashing von Zeichenketten

Hashfunktion aus frühen BerkeleyDB/SDBM:

```
uint32 hash(String str)
{
    uint32 h = 0;
    for (int i = 0; i < str.size(); ++i)
        h = h * 65599 + str[i];
   return h;
}
```

Als Bitoperationen:

```
h = (h << 6) + (h << 16) - h + str[i];
```

Moderne Hashfunktionen

Fowler–Noll–Vo Hashfunktion (DNS-Server, Databases)

```
unsigned int hash(String str)
{
    unsigned int h = offset;
    for (int i = 0; i < str.size(); ++i)
    {
        h = h * prime;
        h = h XOR str[i];
    }
    return h;
}
```

Für 32-bit: offset = 2166136261, prime = 16777619.
Für 64-bit: offset = 14695981039346656037, prime = 109951162821 1.

Noch aktueller: MurmerHash (Perl, Hadoop, etc)