Programmiertechnik II

Ralf Herbrich

Hashing

Überblick

1. Grundlagen
2. Hashfunktionen
3. Kollisionen
   - *Overflow Hashing*
   - Sondieren
4. Bloom-Filter

Überblick

1. **Grundlagen**
2. Hashfunktionen
3. Kollisionen
   - *Overflow Hashing*
   - Sondieren
4. Bloom-Filter

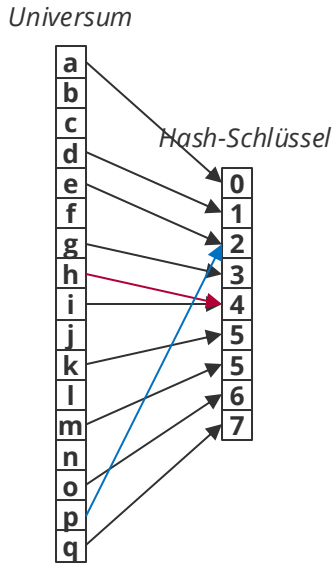
Wörterbücher

- **Wörterbuch**: Verwalte Liste S aus $|S| = n$ Wörtern
  - **Drei Operationen** sollen unterstützt werden
    - Speichere Wort in A
    - Prüfe ob ein Wort in A ist
    - Lösche Wort von A
- **Idee 1**: Speichere Worte in einer verketteten Liste
  - **Vorteil**: Speichern & Löschen in $O(1)$
  - **Nachteil**: Prüfe ob ein Wort in Liste ist: $O(n)$
- **Idee 2**: Speichere Worte in einem Feld mit eindeutigem Index $\sum_i w_i \cdot 26^i$
  - **Vorteil**: Speichern, Prüfen und Löschen in $O(1)$!
  - **Nachteil**: Array wächst exponentiell mit Länge der Worte und ist so gut wie leer
    - Durchschnittliche Länge von Worten in Deutsch: $10.6 \approx 11$ Buchstaben
    - $|A| = 95{,}428{,}956{,}661{,}682{,}176$ vs. $500{,}000$ deutschen Worten (0.000000013%)

Hashing

- **Idee**: Benutze eine **Hashfunktion** $h: U \to \{0, \dots, M-1\}$ um für einen Schlüssel $s \in U$ eine Speicheradresse $h(s)$ in ein Array A zu berechnen und speichere den Wert an der Stelle $A[h(s)]$

- **Eigenschaften** von Hashfunktionen:
  - Jedes Paar $s_1, s_2 \in U$ mit $s_1 \neq s_2$ und $h(s_1) = h(s_2)$ ist eine **Kollision**.
  - h ist **perfekt**, falls sie niemals Kollisionen verursacht.
  - h ist **gleichverteilt** („ideal“) falls $P(h(k) = i) = \frac{1}{M}$ für alle $i \in \{0, \dots, M-1\}$.
  - h ist **ordnungserhaltend** genau dann wenn $s_1 < s_2 \Rightarrow h(s_1) < h(s_2)$.
  - Hashfunktionen sollten **effizient berechenbar** sein
    - Zum Beispiel nicht: Sortiere U und nutze Rang als Hashwert (wäre aber perfekt!)
  - Hashfunktionen sollten **speichereffizient** sein (M sollte so klein wie möglich sein)



Hashing ohne Hashtabelle

- Unix wendet geheime Hashfunktion an um Passwörter zu speichern
- **Speicherung**: Passwörter werden gehashed als Klartext gespeichert:

$$(\text{userid}, h(\text{password}))$$

- **Authentifizierung**: Für gegebenes Passwort p, prüfe ob $h(p) = h(\text{password})$
- **Vorteil**: h ist unabhängig von Nutzer; keine geheimen Informationen werden gespeichert
- **Aber**: Kollisionen sollten unbedingt vermieden werden.

Facebook stored millions of Instagram passwords in plain text

A lot more than initially stated

By Jacob Kastrenakes | @jake_k | Apr 18, 2019, 3:02pm EDT

https://www.theverge.com/2019/4/18/18485599/facebook-instagram-passwords-plain-text-millions-users

Überblick

1. Grundlagen
2. **Hashfunktionen**
3. Kollisionen
   - *Overflow Hashing*
   - Sondieren
4. Bloom-Filter

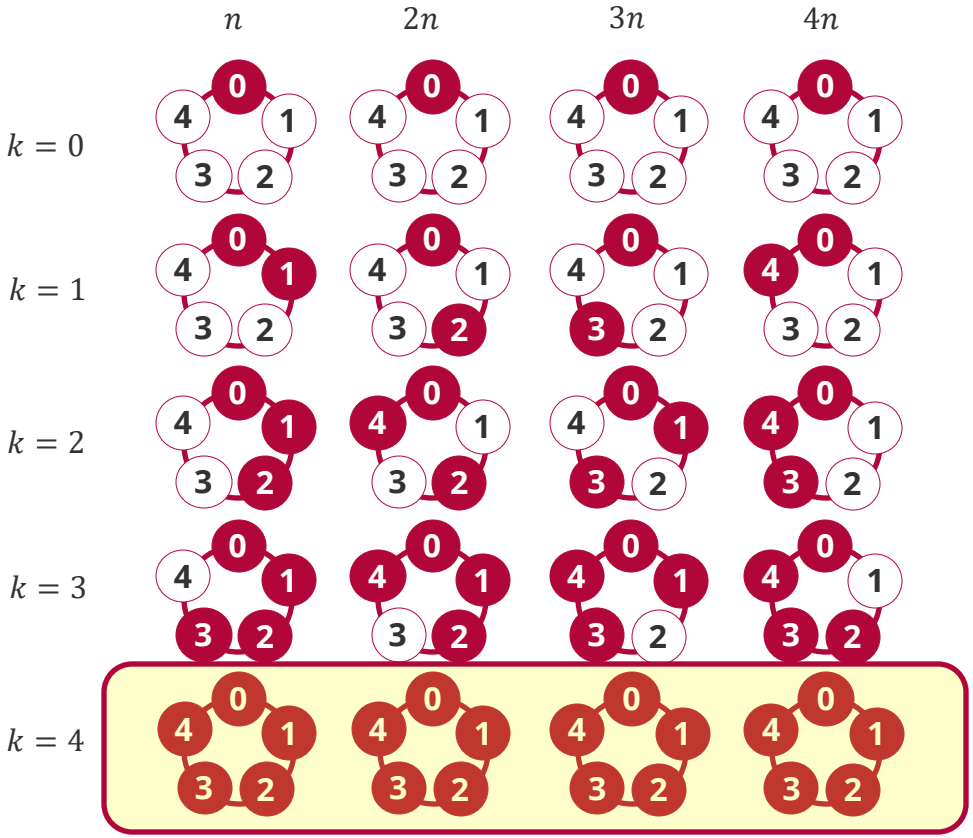
Hashfunktionen: Modulares Hashing

- **Definition**: Falls $U \subseteq \mathbb{N}$ (d.h., s eine Zahl ist), ist eine einfache und überraschend gute Hashfunktion

$$h_{\text{mod}-M}(s) \coloneqq s \bmod M$$

- **Frage**: Wie soll man M wählen?
  - **Fall $M = 2^i$**:
    - Identisch zur logischen Konjunktion des Bitstrings von s mit $2^i - 1$
    - Kopiert nur die untersten i Bits
    - Muster der unteren i Bits bleiben erhalten und alle höheren Bits werden ignoriert ➔ schlechter Hash!
  - **Fall M ist eine Primzahl**:
    - Wenig Kollisionen in vielen Fällen in der Praxis (empirisch belegt)

Primes Modulares Hashing: Intuition ($M = 5$)



Hash-Funktion

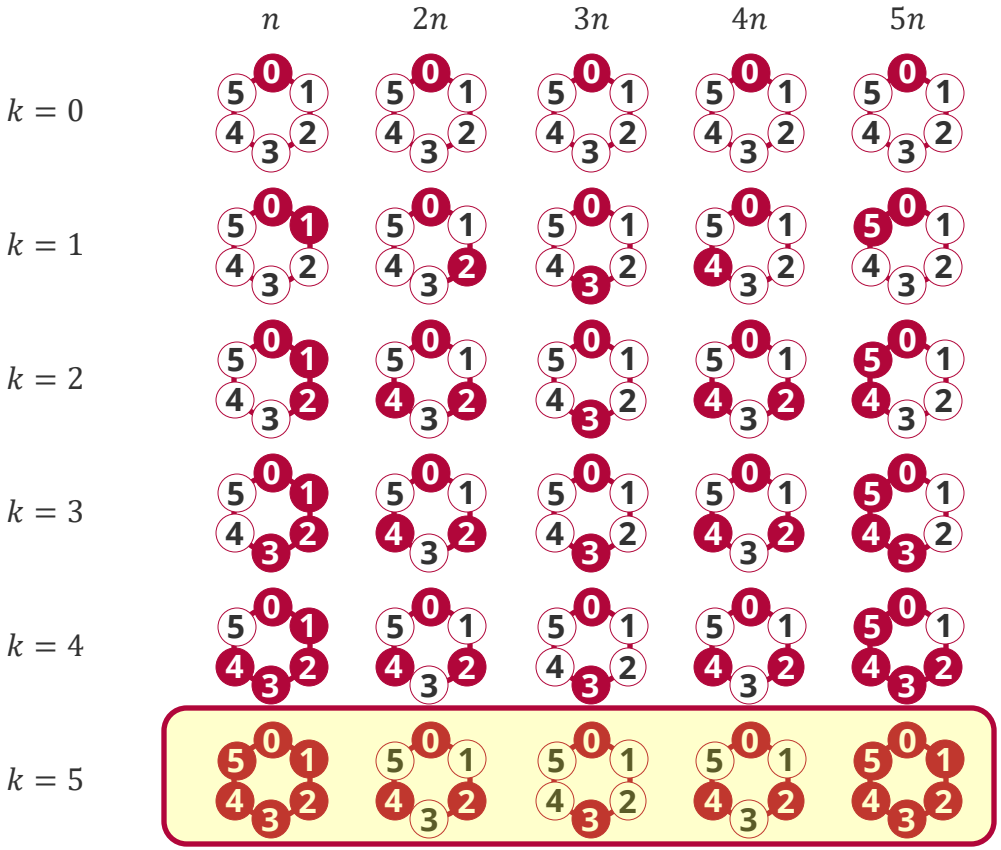
$$h_{\text{mod}-M}(s) \coloneqq s \bmod M$$

Daten-Muster

$$s_k = \{k \cdot n | n \in \mathbb{N}\}$$

Gleichverteilung der Hash-Keys für alle Daten!

Primes Modulares Hashing: Intuition ($M = 6$)



Hash-Funktion

$$h_{\text{mod}-M}(s) := s \bmod M$$

Daten-Muster

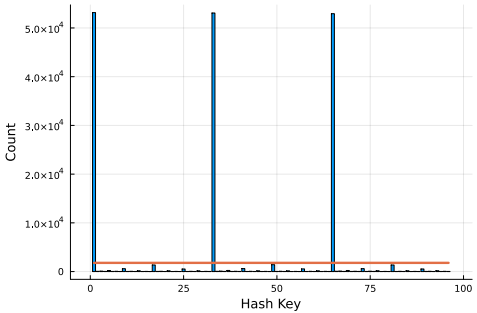
$$s_k = \{k \cdot n | n \in \mathbb{N}\}$$

Keine Gleichverteilung der Hash-Keys für Daten!

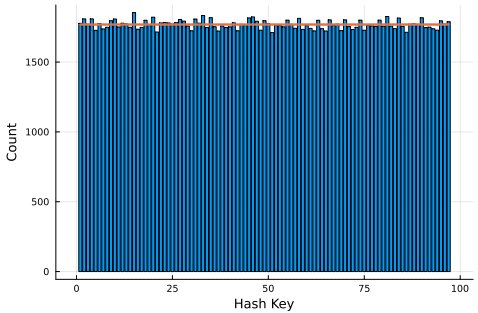
Modulares Hashing: Beispiel

- **Datensatz**: 172,754 weltweiten IPv4 Addressen im Format $aaa.bbb.ccc.ddd$
- **Universum**: Berechen eindeutige IPv4 mittels $a \cdot 256^3 + b \cdot 256^2 + c \cdot 256 + d$

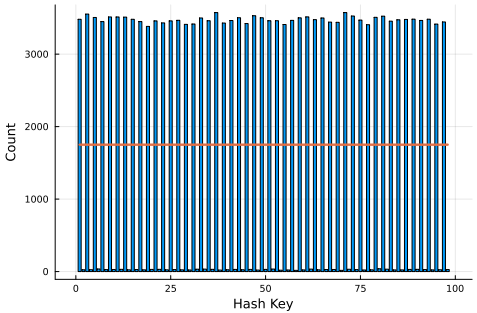
$M = 96$



$M = 97$



$M = 98$



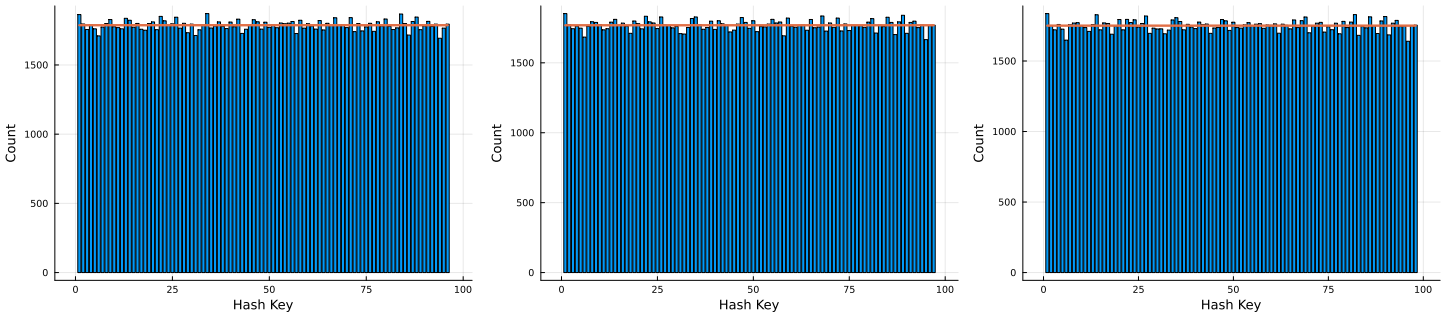
Hashfunktionen: Multiplikative Methode

- **Problem:** Modulares Hashing erzwingt Tabellengröße, die eine Primzahl ist.
- **Idee**: Benutze eine irrationale Zahl $a \in (0,1)$ die eine lange Periodizität hat

$$h_{\text{mult}-M}(s) \coloneqq \lfloor M \cdot (s \cdot a - \lfloor s \cdot a \rfloor) \rfloor$$

- **Intuition**: $s \cdot a - \lfloor s \cdot a \rfloor$ agiert wie ein (Pseudo)-Zufallszahlengenerator in $(0,1)$
- **„Goldner Schnitt“**: $a = \frac{\sqrt{5}-1}{2} \approx 0.618$
- **Beispiel**:

$M = 96$ $M = 97$ $M = 98$



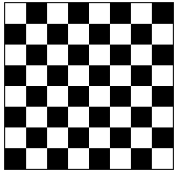
Hashfunktionen für Zeichenketten

- **Grundidee**: Verwende Codewerte der beteiligten Zeichen um einen Ganzzahl zu berechnen
- **Idee 1**: Summe der ASCII oder Uni-Codes
  - **Problem**: $h(\text{“reise”}) = h(\text{“riese”}) = h(\text{“serie”})$
- **Idee 2**: Wichtung nach Position in der Zeichenkette
  - $c_0 R^{k-1} + c_1 R^{k-2} + \cdots + c_{k-1} = c_{n-1} + R(c_{k-2} + R(c_{k-3} + R(\ldots + (c_0) \ldots)))$
  - Schnell zu berechnen (linear in Länge k der Zeichenkette: 1 Multiplikation & Addition)
- **Idee 3 (Zobrist Hash)**: Wenn Zeichenketten gleich lang, generiere Zufallscode $z_{c,i}$ für jedes Zeichen c in jeder Position i und berechne ($\oplus$ ist XOR)

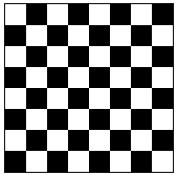
$$h_{\text{zobrist}}(s) \coloneqq z_{s_0,0} \oplus z_{s_1,1} \oplus \cdots \oplus z_{s_k,k}$$

  - Wird in Computerbrettspielen benutzt, um die Stellung mit 64-bit zu speichern
  - Erfunden von Albert Zobrist in 1970 (MIT, heute: Senior Information Scientist at JPL)
  - Wird auch extensiv in AlphaGo benutzt!

$z_{\text{king},0}, \ldots, z_{\text{king},63}$



$z_{\text{queen},0}, \ldots, z_{\text{queen},63}$

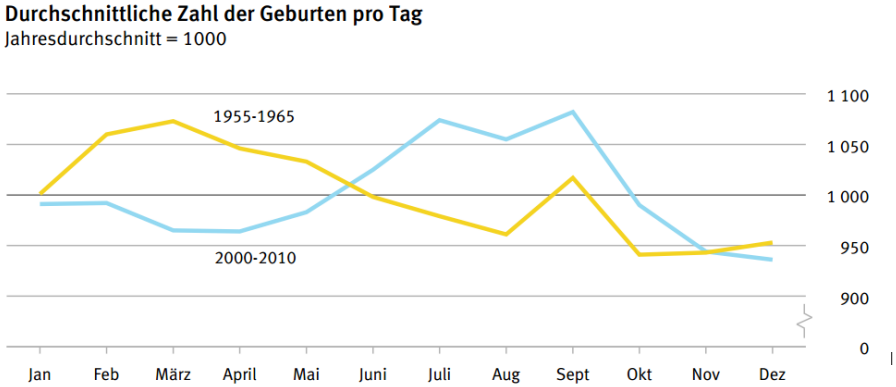


Überblick

1. Grundlagen
2. Hashfunktionen
3. **Kollisionen**
   - *Overflow Hashing*
   - Sondieren
4. Bloom-Filter

Sind Kollisionen überhaupt ein Problem?

- **Annahme**: Gleichverteilte Hashfunktion
  - Bildet beliebigen Schlüssel s auf jede Position $\{0, \dots, M-1\}$ mit gleicher Wahrscheinlichkeit ab
- **Frage**: Gegeben Schlüsselanzahl $n = |S|$ und Anzahl Hash-Keys M: Wie hoch ist die Wahrscheinlichkeit für Kollisionen?
- **Geburtstagsparadoxon**: Gegeben eine Schulklasse, wie wahrscheinlich ist es dass zwei Personen am gleichen Tag Geburtstag haben?
  - Jede Person ist ein Schlüsselwert
  - Jeder Tag eines Jahres ist ein Hash-Key ($M = 365$)
  - **Annahme**: Jeder Tag gleich wahrscheinlich als Geburtstag eines einzelnen



Quelle: Statistisches Bundesamt

2023 Calendar

January

S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

February

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28				

March

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

April

S	M	T	W	T	F	S
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30						

May

S	M	T	W	T	F	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

June

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

July

S	M	T	W	T	F	S
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

August

S	M	T	W	T	F	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

September

S	M	T	W	T	F	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

October

S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

November

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30		

December

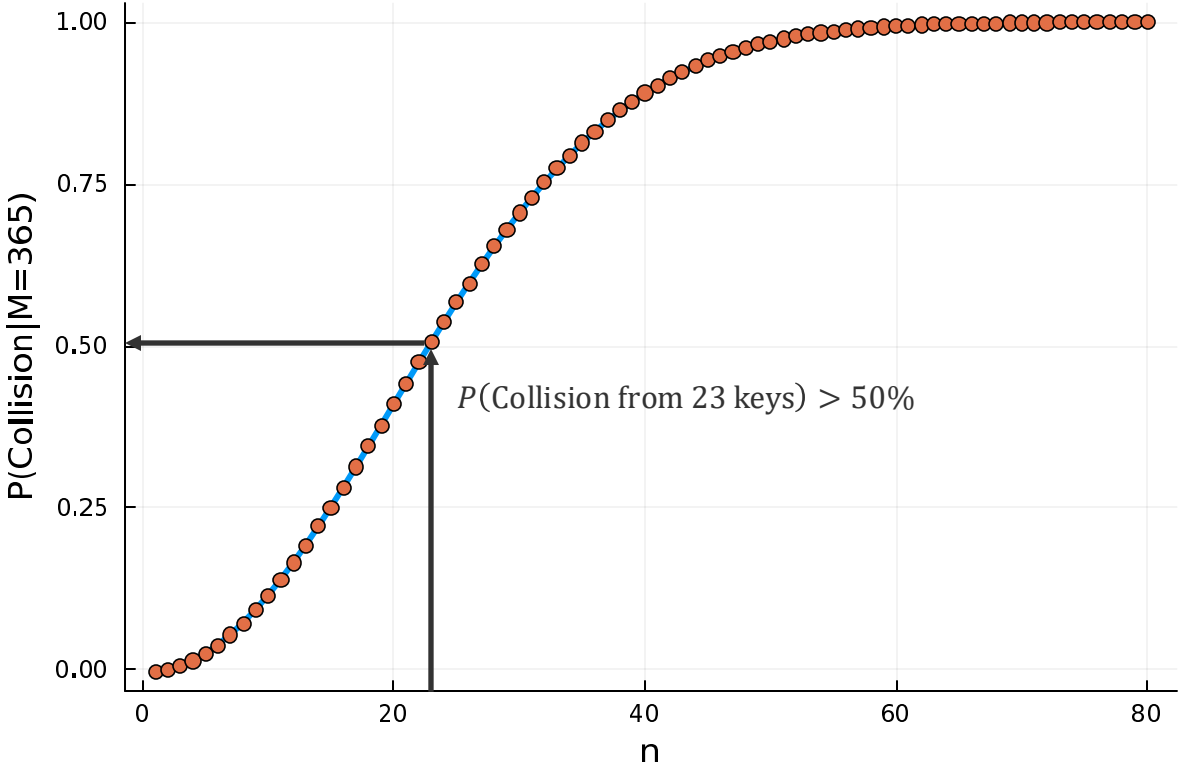
S	M	T	W	T	F	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

Geburtstagsparadoxon

- **Wahrscheinlichkeitsmodell**: Urne mit $M = 365$ Kugeln: n mal ziehen mit Zurücklegen
  - $p(n, M) =$ Wahrscheinlichkeit, eine Kugel mindestens zweimal zu ziehen?
- **Lösungsansatz**: Komplement $q(n, M) = 1 - p(n, M)$ betrachten
  - Komplement: Keine Kugel mehr als einmal zu ziehen
- **Lösung**
  - Alle Kugeln sind blau. Jede gezogene Kugel wird rot gefärbt vor dem Zurücklegen.
  - Wahrscheinlichkeit dass die erste Kugel blau ist: $\frac{M}{M} = 1$
  - Wahrscheinlichkeit dass die zweite Kugel blau ist: $\frac{M-1}{M}$
  - Wahrscheinlichkeit dass die dritte Kugel blau ist: $\frac{M-2}{M}$

$$p(n, M) = 1 - q(n, M) = 1 - \left(\prod_{i=1}^{n} \frac{M - (i-1)}{M} \right) = 1 - \frac{M!}{(M-n)! \cdot M^n}$$

Geburtstagsparadox (ctd)



Verfahren zur Kollisionsbehandlung

1. ***Overflow Hashing*** (*chaining*, verketten): Kollisionen werden **außerhalb** vom Feld A mit M Elementen gespeichert.
   - Zusätzlicher Speicherplatz nötig
   - Feld hat feste Größe und muss nicht geändert werden
   - Kann „beliebig“ viele Daten aufnehmen
2. ***Closed Hashing*** (*probing*, sondieren): Kollisionen werden **innerhalb** vom Feld A mit M Elementen gespeichert
   - Kein zusätzlicher Speicher nötig
   - M ist obere Schranke für Schlüsselanzahl, die gespeichert werden kann
3. **Dynamisches Hashing**: Kollisionen werden **innerhalb** vom Feld A gespeichert, welches seine Größe verändern kann
   - Das Feld selbst kann wachsen (und schrumpfen)
   - Wird hier nicht vertieft (sondern in Datenbanksysteme II)

Überblick

1. Grundlagen
2. Hashfunktionen
3. Kollisionen
   - ***Overflow Hashing***
   - Sondieren
4. Bloom-Filter

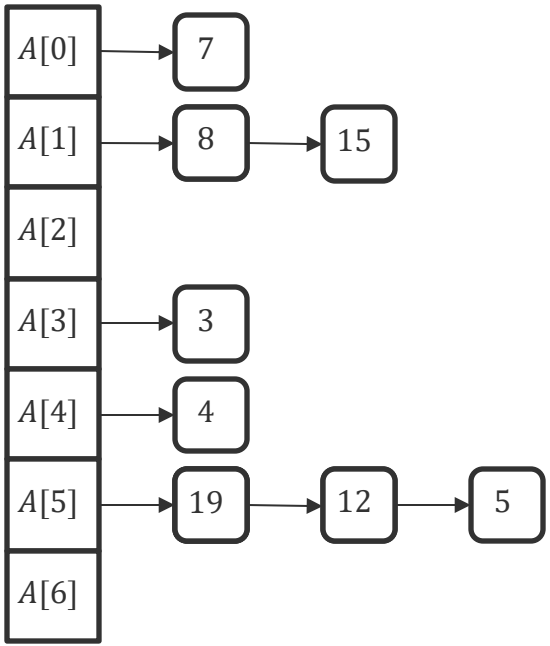
Overflow Hashing

- **Idee**: Feld A speichert nicht die Schlüssel sondern den Zeiger auf separate Speicherstrukturen (Listen).
- **Vorteil**: M kann klein(er) sein, bzw. Wahl nicht entscheidend
- **Zwei Varianten**
  1. Separate Verkettung (*separate chaining*)
     - $A[i]$ speichert Tupel (k, p)
     - k ist der initiale Schlüssel, der auf i gehashed wurde (d.h. $i = h(k)$)
     - p ist Zeiger auf eine Liste, in der alle Schlüssel gespeichert sind, die auf i gehashed wurden, *außer dem ersten Schlüssel*.
     - Gut, falls wenige Kollisionen und kleine Schlüssel
  2. Direkte Verkettung (*sequential chaining*)
     - $A[i]$ ist Zeiger auf Liste, in der *alle* Schlüssel gespeichert sind, die auf i gehashed wurden.
     - Effizienter bei häufigen Kollisionen

```
// Implements the class for a symbol table based on separate-chaining hash table
template <typename Key, typename Value>
class SeparateChainingHashST : public ST<Key, Value> {
    int n;                                  // number of key-value pairs
    int m;                                  // hash-table size
    SequentialSearchST<Key, Value>* table;  // array of linked-list symbol tables
    std::hash<Key> key_hash;                // hashing function for the key
}
```

Beispiel: *Overflow Hashing* mit Direkter Verkettung

- $h(s) = s \bmod 7$, Einfügen am Listenkopf, $S = \{5,15,3,7,8,4,12,19\}$




```
// put a key-value pair into the hash table
void put(const Key& key, const Value& val) {
    int i = hash(key);
    if (!table[i].contains(key)) n++;
    table[i].put(key, val);
    return;
}
```

hash_st.h

Average Case Komplexität

- **Annahme**: Sei h gleichverteilt.
- **Füllgrad**: Nach Einfügen von n Schlüsseln hat jede *overflow* Liste $\alpha \approx n/M$ Elemente. α ist der *Füllgrad* der Hashtabelle.
- **Frage**: Wie viele Vergleiche benötigt die $(n+1)$-te Operation durchschnittlich?
  - Einfügen: $O(1)$
  - Suchen: Falls Schüssel enthalten: $\alpha/2$ Vergleiche; sonst α Vergleiche
  - Löschen: $\alpha/2$ Vergleiche
- **Beobachtung**: Suche mit durchschnittlich $\alpha/2$ Vergleichen nicht günstig?
  - **Aber**: Typischer Anwendungsfall hat sehr kleines α (α<1 – Suche in konstanter Zeit!)
  - **Beispiel**: $|S| = n = 10.000.000$ und $M = 1.000.000$
    - Hashtabelle: ≈ 5 Vergleiche
    - Sortiertes Array: $\log(10.000.000) \approx 23$ Vergleiche

Überblick

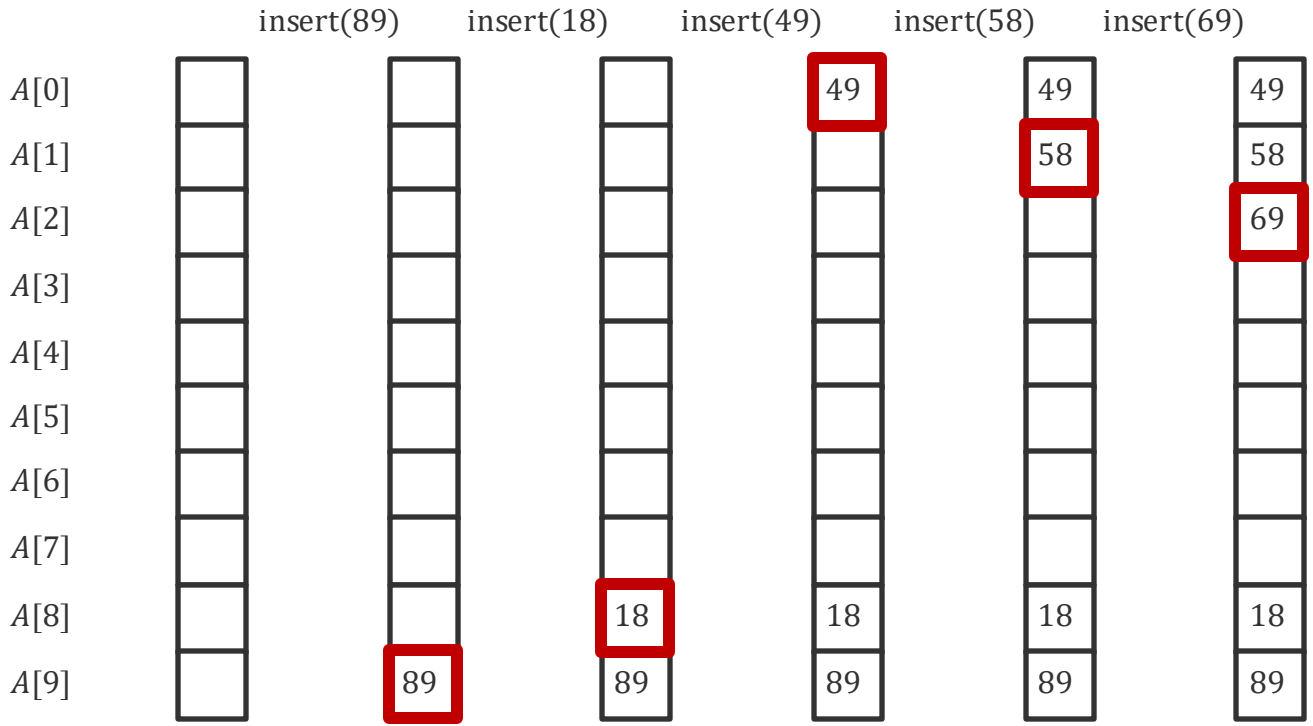
1. Grundlagen
2. Hashfunktionen
3. Kollisionen
   - *Overflow Hashing*
   - **Sondieren**
4. Bloom-Filter

Lineares Sondieren

- **Problem**: *Overflow Hashing* verwendet zusätzlichen Speicher.
- **Idee**: Verwende (nächste) freie Stellen in Hashtabelle
  - Falls $A[h(s)]$ besetzt, versuche $A[h(s)+1], A[h(s)+2], \dots, A[h(s)+i], \dots$
  - **Genauer**: $\{A[(h(s)+i) \bmod M] \mid i = 1,2,\dots,M-1\}$
- **Varianten**:
  - $\{A[(h(s)+2\cdot i) \bmod M] \mid i = 1,2,\dots,M-1\}$
  - $\left\{A\left[\left(h(s)+(-1)^{i+1}\cdot\left\lfloor\frac{i+1}{2}\right\rfloor\right) \bmod M\right] \mid i = 1,2,\dots,M-1\right\}$
- **Suche**: Falls $s \neq A[h(s)]$ suche weiter (je nach Variante) bis Schlüssel gefunden oder bis leere Stelle erreicht
- **Löschen**: Besetzte Elemente auf dem Sondierungspfad werden gelöscht!
  - **Lösung**: Spezielles Symbol hinterlassen

Beispiel: Lineares Sondieren

- $h(s) = s \bmod 10$



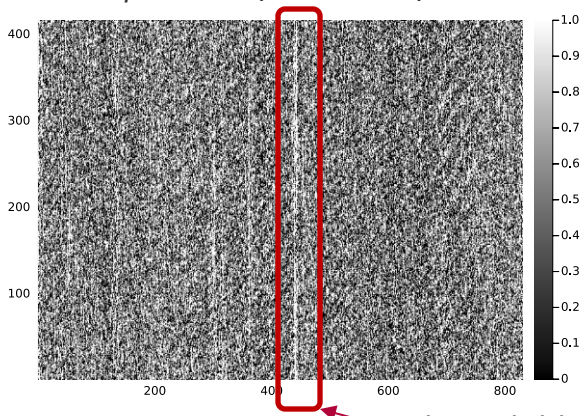
Analyse: Lineares Sondieren

- **Satz (Knuth, 1962)**: In einer auf linearer Sondierung basierenden Hashtabelle mit M Einträgen und $N = \alpha M$ Schlüsseln ist die durchschnittliche Anzahl von Sondierungen, β, für eine gleichverteilte Hashfunktion

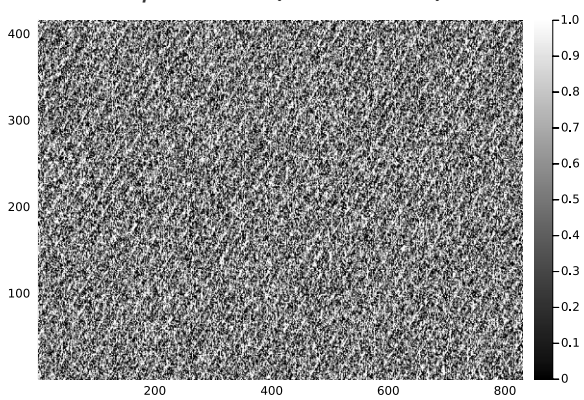
$$\frac{1}{2} \cdot \left(1 + \frac{1}{1-\alpha}\right)$$

- **Beispiel**: Lineare Sondierung mit 172,754 weltweiten IPv4 Addressen

$\beta = 1.67$ (mod Hash)



$\beta = 1.56$ (mult Hash)



Überblick

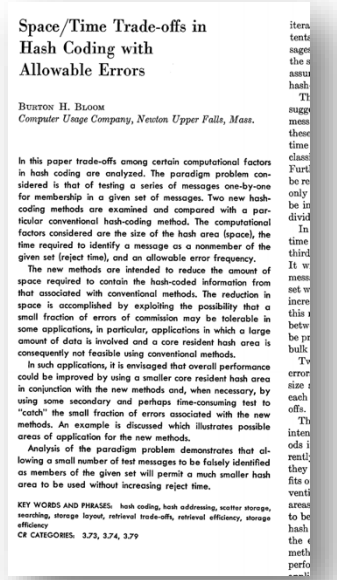
1. Grundlagen
2. Hashfunktionen
3. Kollisionen
   - *Overflow Hashing*
   - Sondieren
4. **Bloom-Filter**

Bloom Filter

- **Ursprüngliches Problem**: Verwalte Liste S aus $|S| = n$ Wörtern mit drei Operationen
  - Speichere Wort in A
  - Prüfe ob ein Wort in A ist
  - Lösche Wort von A

Einfacheres Problem!

- **Idee (Bloom 1970)**: Baue Hashtabelle $A \in \{0,1\}$ und nutze A um zu markieren ob ein Schlüssel in S ist oder nicht.
  - **Initialisiere** A: $\forall\, s \in S: A[h(s)] = 1$, sonst $\forall\, i \in dom(A) \setminus \{h(s') \mid s' \in S\}: A[i] = 0$
  - **Suche nach Schlüssel** s
    - Falls $A[h(s)] = 0$, wissen wir *sicher* dass $s \notin S$!
    - Falls $A[h(s)] = 1$, wissen wir *nicht sicher* dass $s \in S$
  - A wirkt als Filter: Ein Bloom Filter



Space/Time Trade-offs in Hash Coding with Allowable Errors

BURTON H. BLOOM
Computer Usage Company, Newton Upper Falls, Mass.

In this paper trade-offs among certain computational factors in hash coding are analyzed. The paradigm problem considered is that of testing a series of messages one-by-one for membership in a given set of messages. Two new hash-coding methods are examined and compared with a particular conventional hash-coding method. The computational factors considered are the size of the hash area (space), the time required to identify a message as a nonmember of the given set (reject time), and an allowable error frequency.

The new methods are intended to reduce the amount of space required to contain the hash-coded information from that associated with conventional methods. The reduction in space is accomplished by exploiting the possibility that a small fraction of errors of commission may be tolerable in some applications, in particular, applications in which a large amount of data is involved and a core resident hash area is consequently not feasible using conventional methods.

In such applications, it is envisaged that overall performance could be improved by using a smaller core resident hash area in conjunction with the new methods and, when necessary, by using some secondary and perhaps time-consuming test to "catch" the small fraction of errors associated with the new methods. An example is discussed which illustrates possible areas of application for the new methods.

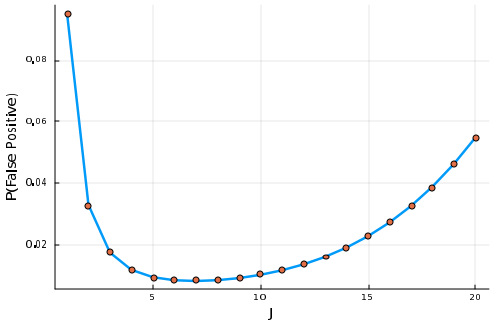
Analysis of the paradigm problem demonstrates that allowing a small number of test messages to be falsely identified as members of the given set will permit a much smaller hash area to be used without increasing reject time.

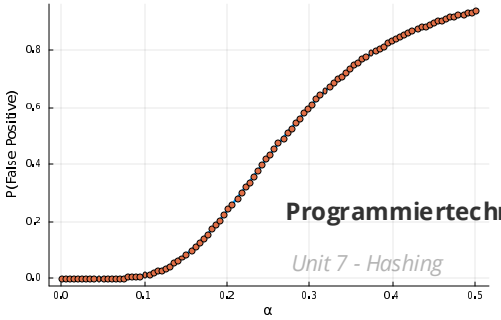
KEY WORDS AND PHRASES: hash coding, hash addressing, scatter storage, searching, storage layout, retrieval trade-offs, retrieval efficiency, storage efficiency
CR CATEGORIES: 3.73, 3.74, 3.79

Bloom Filter: Verbesserung

- **Bloom's Idee**: Wähle J unabhängige, gleichverteilte Hashfunktionen $h_1, \dots, h_J$
  - **Unabhängig**: Die Werte der einen Hashfunktion sind statistisch unabhängig von den Werten aller anderen Hashfunktionen.
- **Initialisiere A**: $\forall\, s \in S \forall j \in \{1, \dots, J\}: A\left[h_j(s)\right] = 1$, sonst $\forall\, i \in dom(A) \setminus \{h(s') |\, s' \in S\}:\ A[i] = 0$
- **Suche nach Schlüssel s**:
  - Falls es ein j' gibt, so dass $A\left[h_{j'}(s)\right] = 0$, wissen wir *sicher* dass $s \notin S$!
  - Falls für alle j: $A\left[h_j(s)\right] = 1$, wissen wir *nicht sicher* dass $k \in S$
- **Falsch-Positiv Rate**: Approximative gegeben durch

$$P(\text{False Positive}) = \left(1 - \exp(-J\alpha)\right)^J$$





Zusammenfassung

- **Grundlagen**
  - Hashing wird benutzt, um Speicherplatz und Komplexität von Suche auszubalancieren (Extreme: Liste & Feld)
  - Größtes Problem: Kollisionen
- **Hashfunktionen**
  - Modulares Hashing ist einfach aber verlangt Primzahlgröße
  - Multiplikative Methode und Zobrist Hashing sind besser
- **Kollisionen**
  - Tritt häufiger auf, als man denkt (Geburtstagsparadox)
  - Zwei Methoden, um Kollisionen zu behandeln
    - *Overflow Hashing* (zusätzlicher Speicher): Jedes Element ist eine Liste
    - Sondieren (kein zusätzlicher Speicher): Fülle nächste Element aus
- **Bloom-Filter**
  - Speichereffizient, wenn nur geprüft werden soll, ob ein Schlüssel enthalten ist

Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Viel Spaß bis zur nächsten Vorlesung!