# 7. Hashing

## Einleitung

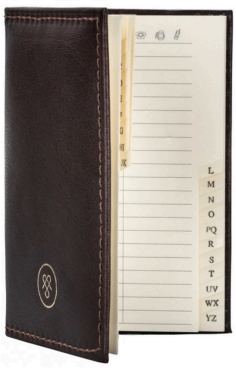
Wie auch sem_2/AlgoDat/vo/6. Suchbäume ist Hashing eine Lösung für das Wörterbuchproblem

### Beispiel

- Vergleich mit einem „Telefonregister“:
  Eine Seite für jeden Anfangsbuchstaben.
- Einfache Berechnung der Position in der Hashtabelle mit der Ordinalzahl $(\mathrm{ord})$ des ersten Buchstabens im Namen $s$ (z.B. $\mathrm{ord}('A') = 0$, $\mathrm{ord}('B') = 1, \ldots$).
- Hashfunktion $h$: hier $h(s) = \mathrm{ord}(s[0])$.

| | |
|---|---|
| 0 ("A") | Anna 123 |
| 1 ("B") | Barbara 222 |
| 2 ("C") | |
| 3 ("D") | Doris 404, Daniel 343 |
| 4 ("E") | |
| 5 ("F") | |
| 6 ("G") | Günther 777 |
| 7 ("H") | |
| ... | ... |
| 25 ("Z") | |

Hashtabelle T



# Grundlagen von Hashing

## Hashtabelle:

- Wir gehen davon aus, dass die Hashtabelle $T$ mit einer vorgegebenen Tabellengröße $m$ als Array mit den Indizes $0, \ldots, m-1$ realisiert wird.
- Für eine Hashtabelle der Größe $m$, die aktuell $n$ Schlüssel speichert, ist $\alpha = \frac{n}{m}$ der Belegungsfaktor der Tabelle.

## Hashfunktion:

- Wir wählen eine Hashfunktion $h : \mathcal{K} \to \{0, \ldots, m-1\}$, die jedem Schlüssel $k \in \mathcal{K}$ einen eindeutigen – aber i.A. nicht umgekehrt eindeutigen – Hashwert zuordnet.

Vorteil: Laufzeit für die Operationen Suchen, Einfügen und Entfernen liegt im Idealfall in $O(1)$, wenn wir annehmen, dass $h(s)$ in konstanter Zeit berechnet werden kann.

## Einschränkung:

- Gilt $h(k) = h(k')$ für $k \neq k'$, d.h., zwei verschiedene Schlüssel haben den gleichen Hashwert, so wird dies Kollision genannt.
- $O(1)$ gilt nur unter der Annahme, dass die Anzahl der Kollisionen vernachlässigbar klein ist.
- Im Erwartungsfall (bei entsprechender Konfiguration) erreichbar.
- Im Worst-Case liegt der Aufwand in $O(n)$.

## Zu klärende Punkte:

- Wie erfolgt die Kollisionsbehandlung?
  - Verkettung der Überläufer
  - Offene Hashverfahren
- Was ist eine gute Hashfunktion?
  - Divisions-Rest-Methode
  - Multiplikationsmethode
- Wie soll die Tabellengröße $m$ gewählt werden?

Alle diese Aspekte beeinflussen die Güte/Effizienz der Hashtabelle.

# Hashfunktionen

Was charakterisiert eine gute Hashfunktion?

Vor allem:

- Verwendete Schlüssel sollen möglichst gleichmäßig auf alle Plätze $0, \ldots, m-1$ der Hashtabelle aufgeteilt werden.
- Auch kleinste Änderungen im Schlüssel sollen zu einem anderen, möglichst unabhängigen Hashwert führen.
- Das Beispiel mit der Ordnungszahl des Anfangsbuchstaben ist i.A. keine gute Wahl.

## Divisions-Rest-Methode

Annahme: $k \in \mathbb{N}$

Berechnung:

$$h(k) = k \mod m$$

Eigenschaften:

- Die Hashfunktion kann sehr schnell berechnet werden.
- Die richtige Wahl von $m$ ist hier sehr wichtig.
- Eine gute Wahl für $m$ ist eine Primzahl.

Schlechte Wahl für $m$:

- $m = 2^p$: Nur die letzten $p$ Binärziffern spielen eine Rolle!
- $m = 10^p$: Analog bei dezimalen Zahlen.
- $m = r^p$: Analog bei $r$-adischen Zahlen.
- Aber auch $m = r^p \pm c$ für kleine $c$ kann problematisch sein:
    - z.B. $m = 2^7 - 1 = 127$. Buchstaben als Zahlen interpretieren
        - h("p") = (112) mod 127 = 112
        - h("t") = (116) mod 127 = 116
        - h("pt") = (112 * 128 + 116) mod 127 = 14452 mod 127 = 101
        - h("tp") = (116 * 128 + 112) mod 127 = 14960 mod 127 = 101 (Schlüssel in denen zwei Buchstaben vertauscht sind haben hier häufig den gleichen Hashwert)

---

## Divisions-Rest-Methode für Strings

Schlüssel: String $s = (s_1, \ldots, s_l)$, wobei $s_i \in \{0, \ldots, 127\}$

Berechnung: $k = 128^{l-1}s_1 + 128^{l-2}s_2 + \ldots + s_l$
Die sehr großen ganzzahligen Werte sind problematisch!

Berechnung mit Horner-Schema:

$$k = (\ldots((s_1 \cdot 128 + s_2) \cdot 128 + s_3) \cdot 128 + \ldots + s_{l-1}) \cdot 128 + s_l$$

Es gilt:

$$k \mod m = (\ldots((s_1 \cdot 128 \mod m + s_2) \mod m \cdot 128 \mod m + s_3) \\ \mod m \cdot 128 \mod m + \ldots + s_{l-1}) \mod m \cdot 128 \mod m + s_l) \mod m$$

Konsequenz: Keine Zwischenresultate $> (m-1) \cdot 128 + 127$. Berechnung mit üblichen Integer-Typen so gut möglich.

---

## Multiplikationsmethode

### Grundlegende Idee:

- Wir nehmen wieder an: Schlüssel $k \in \mathbb{N}$
- Gegeben: irrationale Zahl $A$
- Berechnung:

$$h(k) = \lfloor m \cdot (k \cdot A \mod 1) \rfloor$$

  wobei $(k \cdot A \mod 1) = k \cdot A - \lfloor k \cdot A \rfloor \in [0, 1)$
- Der Schlüssel wird mit $A$ multipliziert, der ganzzahlige Anteil des Resultats wird abgeschnitten.
- Man erhält einen Wert in $[0, 1)$, dieser wird mit der Tabellengröße $m$ multipliziert, das Ergebnis gerundet.
- Eigenschaft: die gleichmäßige Streuung bestätigt;
  - Für eine Schlüsselmenge $1, 2, 3, \ldots$ liegt $i \cdot A \mod 1$ – bis auf den Faktor $1/m$ – im gleichen Intervall zwischen zuvor ermittelten Werten, 0 und 1.

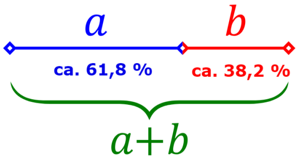
### Wahl für $A$

Allgemein:

- Die Wahl von $m$ ist hier unkritisch, sofern $A$ eine irrationale Zahl ist.

Beste Wahl für $A$: der goldene Schnitt

$$A = \Phi^{-1} = \frac{\sqrt{5} - 1}{2} \approx 0.6180339887\ldots$$



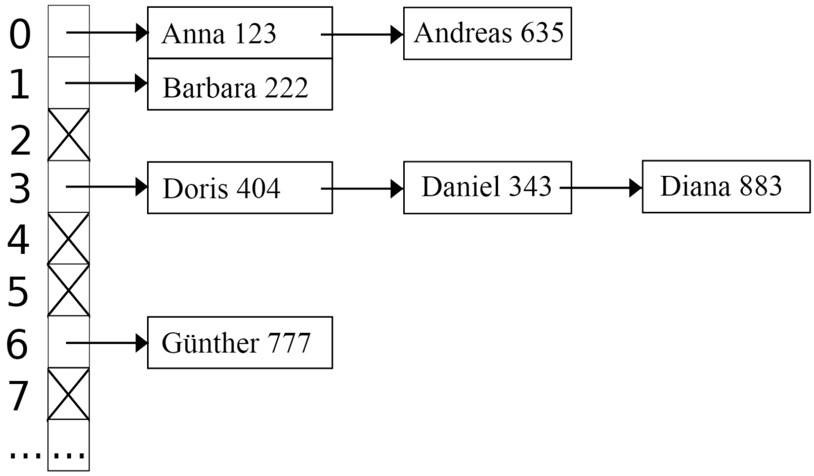

$$\frac{a}{a+b} = \frac{b}{a}$$

- *Eine Begründung warum $\Phi^{-1}$ die beste Wahl ist findet sich in: D.E. Knuth: The Art of Computer Programming, Vol.3: Sorting and Searching, Addison-Wesley, 1973.*

# Kollisionsbehandlung, Verkettung der Überläufer

## Idee

Idee: Jedes Element der Hashtabelle ist eine verkettete Liste.




## Initialisierung

Eingabe: Hashtabelle $T$ = Array von $m$ Verweisen auf die jeweils ersten Elemente.
Ergebnis: Initialisierte Hashtabelle.

```
Initialize(T, m):
for i ← 0 bis m − 1
    T[i] = null
```

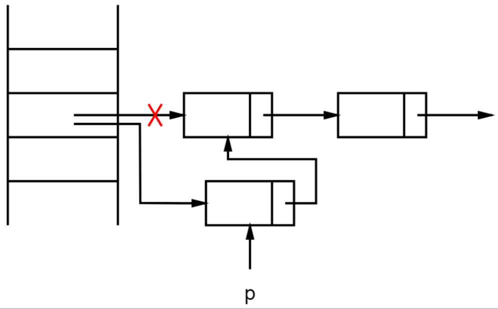
Erläuterung:

- Es wird ein Array $T$ der Größe $m$ erstellt.
- Jedes Element $T[i]$ dieses Arrays wird auf `null` gesetzt, was bedeutet, dass zu Beginn keine Elemente in der Hashtabelle vorhanden sind.
- $m$ repräsentiert die Größe der Hashtabelle oder die Anzahl der Buckets.

## Einfügen

Eingabe: Hashtabelle $T$ und einzufügendes Element $p$.
Ergebnis: Hashtabelle $T$ mit neu eingetragenem Element $p$.

```
Insert(T, p):
pos ← h(p.key)
p.next ← T[pos]
T[pos] ← p
```




Erläuterung:

1. Der Hashwert des Schlüssels von Element $p$ ($p.key$) wird mit der Hashfunktion $h$ berechnet. Dieser Hashwert ($pos$) bestimmt den Index im Array $T$, an dem das Element eingefügt werden soll.
2. $p.next \leftarrow T[pos]$: Das `next`-Feld des einzufügenden Elements $p$ wird auf das aktuelle Element gesetzt, das sich an der berechneten Position $T[pos]$ befindet. Dies ist wichtig für die Kollisionsbehandlung durch Verkettung. Wenn der Bucket leer ist, wird $p.next$ auf `null` gesetzt.
3. $T[pos] \leftarrow p$: Der Zeiger im Array $T$ an der Position $pos$ wird nun auf das neu eingefügte Element $p$ gesetzt. Dadurch wird $p$ zum ersten Element in der Liste (oder dem einzigen Element, falls der Bucket vorher leer war) an dieser Position.

Hinweis: Der Hashwert (und damit die Position in der Hashtabelle) für $p.key$ wird mit der Funktion $h$ berechnet.

## Suchen

Eingabe: Hashtabelle $T$ und gesuchter Schlüssel $k$.
Rückgabewert: Gesuchtes Element $p$.

```
Search(T, k):
p =T[h(k)]
while p ≠ null und p.key ≠ k
    p ← p.next
return p
```

Erläuterung:

1. Der Hashwert des gesuchten Schlüssels $k$ wird mit der Hashfunktion $h$ berechnet. Dies ergibt die Startposition $T[h(k)]$ in der Hashtabelle, an der sich das gesuchte Element befinden könnte.
2. $p \leftarrow T[h(k)]$: Der Zeiger $p$ wird auf das erste Element in der Liste an der berechneten Position gesetzt.
3. while $p \neq null$ und $p.key \neq k$: Es wird so lange durch die verkettete Liste gegangen, wie der aktuelle Zeiger $p$ nicht `null` ist (also das Ende der Liste noch nicht erreicht wurde)

und der Schlüssel des aktuellen Elements ($p.key$) nicht mit dem gesuchten Schlüssel $k$ übereinstimmt.

4. $p \leftarrow p.next$: Wenn der Schlüssel des aktuellen Elements nicht der gesuchte Schlüssel ist, wird $p$ zum nächsten Element in der verketteten Liste gesetzt.

## Entfernen eines Elements aus einer Hashtabelle

Eingabe: Hashtabelle $T$ und Schlüssel $k$ des zu entfernenden Elements (Annahme: Element mit Schlüssel $k$ ist in $T$ enthalten).

Ergebnis: Element mit dem Schlüssel $k$ wurde aus $T$ entfernt.

Algorithmus `Remove(T, k)`:

1. Berechne den Hashwert des Schlüssels: $pos \leftarrow h(k)$
2. Initialisiere $q$ als `null` (Zeiger auf den vorherigen Knoten).
3. Setze $p$ auf den Kopf der Liste an der Hash-Position: $p \leftarrow T[pos]$
4. Suche nach dem Element:

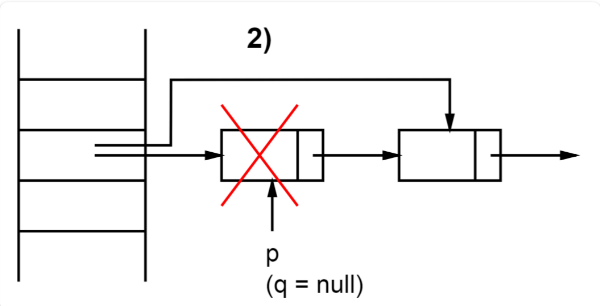
```
while p.key ≠ k:
    q ← p
    p ← p.next
```

- Die Schleife iteriert durch die verkettete Liste an der Position $pos$, bis das Element mit dem Schlüssel $k$ gefunden wird.
- $q$ folgt $p$ und zeigt immer auf den vorherigen Knoten.

5. Fall 1: Element am Kopf der Liste ($q$ ist `null`)

```
if q == null:
    T[pos] ← T[pos].next
```

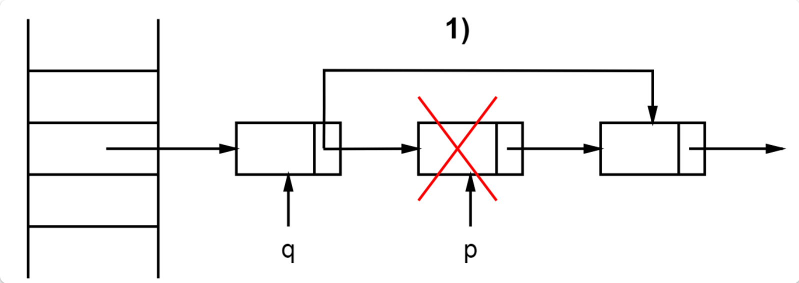
- Wenn $q$ `null` ist, bedeutet das, dass das zu entfernende Element das erste Element in der Liste an der Position $pos$ ist.
- In diesem Fall wird der Zeiger in der Hashtabelle direkt auf das nächste Element gesetzt.




6. Fall 2: Element nicht am Kopf der Liste ($q$ ist nicht `null`)

```
else:
    q.next ← p.next
```

- Wenn $q$ nicht `null` ist, bedeutet das, dass das zu entfernende Element nicht das erste ist.
- Der `next` -Zeiger des vorherigen Elements ($q$) wird auf das nächste Element nach dem zu entfernenden Element ($p$) gesetzt. Dadurch wird $p$ aus der Liste entfernt.




Remove(T, $k$):
$pos \leftarrow h(k)$
$q \leftarrow null$
$p \leftarrow$ T[$pos$]
**while** $p.key \neq k$
    $q \leftarrow p$
    $p \leftarrow p.next$
**if** $q = null$
    T[$pos$] $\leftarrow$ T[$pos$].$next$
**else**
    $q.next \leftarrow p.next;$

# Kollisionsbehandlung, Offene Hashverfahren

## Grundlegende Idee

- Alle Datensätze werden direkt in einem einfachen Array gespeichert.
- Pro Platz ein Flag $f_i \in \{\text{frei}, \text{besetzt}, \text{wieder frei}\}$.

Kollisionsbehandlung: Wenn ein Platz belegt ist, werden weitere Reihenfolge weitere Plätze betrachtet (Sondieren).

Am Anfang sind alle Plätze frei:

| | | |
|---|---|---|
| 0 ("A") | | frei |
| 1 ("B") | | frei |
| 2 ("C") | | frei |
| 3 ("D") | | frei |
| 4 ("E") | | frei |
| 5 ("F") | | frei |
| 6 ("G") | | frei |
| 7 ("H") | | frei |
| ... | | frei |
| 25 ("Z") | | frei |

- Hashfunktion: Ordinalzahl des ersten Buchstabens im Namen
- Sondierreihenfolge: einfach die nächste Position

| | | |
|---|---|---|
| 0 ("A") | Anna 123 | besetzt |
| 1 ("B") | Albert 127 | besetzt |
| 2 ("C") | | frei |
| 3 ("D") | Doris 404 | besetzt |
| 4 ("E") | | frei |
| 5 ("F") | | frei |
| 6 ("G") | Günther 777 | besetzt |
| 7 ("H") | | frei |
| ... | | frei |
| 25 ("Z") | | frei |

- Anna wurde vor Albert eingetragen, Albert kommt daher auf die nächste freie Position 1.
- Ein weiterer Eintrag für Andreas würde an Position 2 gespeichert werden, Armin danach an Position 4, usw.

- Entfernen: Flag wird auf `wieder frei` gesetzt.
  - Im Beispiel wird Anna entfernt.
- Würde nach $h(\text{"Albert"}) = 1$ gesucht, würde Albert nicht mehr gefunden werden, da die Sondierung bei Position 0 abbricht!
- Ein neuer Eintrag für Andreas würde wieder an Position 0 gespeichert werden.

| | | |
|---|---|---|
| 0 ("A") | Anna 123 | wieder frei |
| 1 ("B") | Albert 127 | besetzt |
| 2 ("C") | | frei |
| 3 ("D") | Doris 404 | besetzt |
| 4 ("E") | | frei |
| 5 ("F") | | frei |
| 6 ("G") | Günther 777 | besetzt |
| 7 ("H") | | frei |
| ... | | frei |
| 25 ("Z") | | frei |

---

## Im Detail

- Alle Elemente werden – im Gegensatz zur Verkettung der Überläufer – direkt im Array gespeichert.
- Zu jedem Platz $i = 0, \ldots, m-1$ wird ein Flag $f_i \in \{\text{frei}, \text{besetzt}, \text{wieder frei}\}$ gespeichert.
- In der anfangs leeren Tabelle gilt für alle $i$: $f_i = \text{frei}$.
- Beim Einfügen wird das neue Element am ersten mit $\text{frei}$ oder $\text{wieder frei}$ markierten Platz eingefügt, das Flag wird auf $\text{besetzt}$ gesetzt.
- Die Suche durchmustert alle Plätze bis der gesuchte Schlüssel entweder gefunden wird oder ein Platz als $\text{frei}$ markiert ist (Schlüssel nicht enthalten).
- Das Entfernen setzt das Flag für das zu entfernende Element auf $\text{wieder frei}$.

---

## Sondierung

Kollisionsbehandlung: Wenn ein Platz belegt ist, so werden in einer bestimmten Reihenfolge weitere Plätze in Betracht gezogen.

Sondierungsreihenfolge (Probing): Reihenfolge der auszuprobierenden Plätze.

$\rightarrow$ Die Hashfunktion wird zu einer Sondierungsfunktion $h(k, i)$ für Schlüssel $k$ und Positionen $i = 0, 1, \ldots, m-1$ erweitert, die Sondierungsreihenfolge ist dann $h(k, 0), h(k, 1), \ldots, h(k, m-1)$.

## Lineares Sondieren

Gegeben: Eine normale Hashfunktion:

$$h' : K \to \{0, 1, \dots, m-1\}$$

Lineares Sondieren: Wir definieren für $i = 0, 1, \dots, m-1$:

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

### Beispiel

Beispiel: $m = 8$, $h'(k) = k \bmod m$

Schlüssel und Wert der Hashfunktion:

| $k$ | 10 | 19 | 31 | 22 | 14 | 16 |
|---|---|---|---|---|---|---|
| $h'(k)$ | 2 | 3 | 7 | 6 | 6 | 0 |

Belegung der Hashtabelle:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 14 | 16 | 10 | 19 | | | 22 | 31 |

Durchschnittliche Zeit: Für eine erfolgreiche Suche $\frac{9}{6} = 1,5$ (siehe Tabelle).

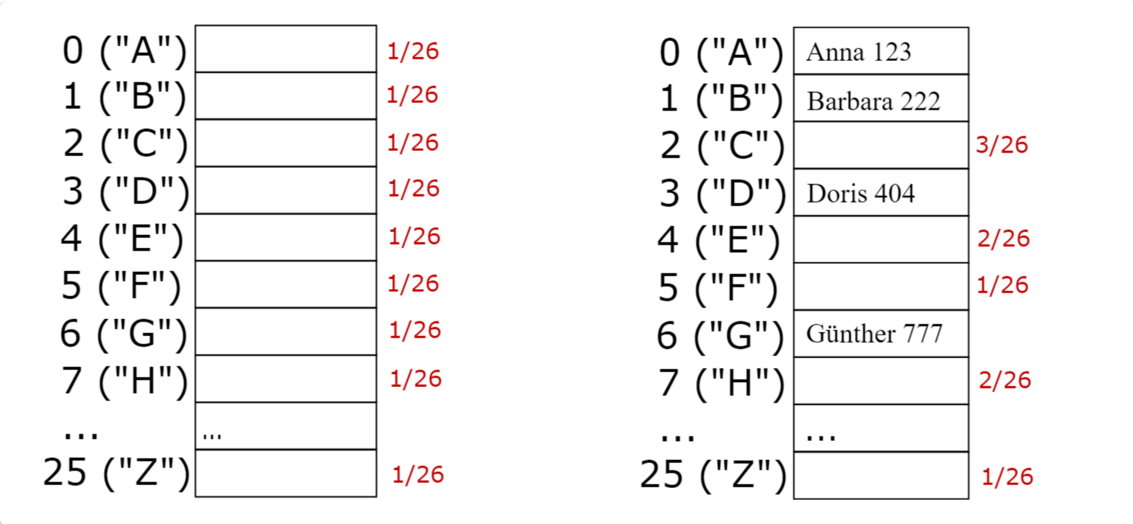
| $k$ | 10 | | 19 | | 31 | | 22 | | 14 | | 16 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | + | 1 | + | 1 | + | 1 | + | 3 | + | 2 | = | 9 |

### Probleme

- Nach dem Einfügen ist die Wahrscheinlichkeit für einen neu einzufügenden Schlüssel in der Hashtabelle an einer gewissen Position gespeichert zu werden für die verschiedenen Positionen unterschiedlich.
- Wird ein Platz belegt, dann verändert sich die Wahrscheinlichkeit für das Einfügen an seinem nachfolgenden Platz.

Beispiel:

- In der leeren Tabelle haben alle Plätze die gleiche Wahrscheinlichkeit.
- Nach dem Einfügen von verschiedenen Schlüsseln verändern sich die Wahrscheinlichkeiten. Z.B. werden auf der rechten Seite im Eintrag $T[2]$ nach dem Einfügen von Anna und Barbara alle Schlüssel $k$ mit $h'(k) = 0$ oder $h'(k) = 1$ oder $h'(k) = 2$ gespeichert; im Eintrag $T[5]$ dagegen nur alle Schlüssel $k$ mit $h'(k) = 5$.




Probleme:

- Lange belegte Teilstücke der Hashtabelle haben eine stärkere Tendenz zu wachsen als kurze.
- Dieser Effekt wird noch verstärkt, weil lange belegte Teilstücke zu größeren Zusammenwachsen, wenn die Lücken zwischen ihnen geschlossen werden.
- Als Folge dieses Phänomens der primären Häufung (primary clustering) verschlechtert sich die Effizienz des linearen Sondierens drastisch, sobald sich der Belegungsfaktor $\alpha$ dem Wert 1 nähert.

---

## Uniform Hashing

- Idealform des Sondierens.
- Jeder Schlüssel erhält mit gleicher Wahrscheinlichkeit eine bestimmte der $m!$ Permutationen von $0, 1, \ldots, m-1$ als Sondierungsreihenfolge zugeordnet.
- Ist in der Praxis schwierig zu implementieren und wird daher mit den nachfolgenden Verfahren approximiert.

---

## Quadratisches Sondieren

Idee: Um die primäre Häufung des linearen Sondierens zu vermeiden, wird beim quadratischen Sondieren für Schlüssel $k$ von $h'(k)$ aus mit quadratisch wachsendem Abstand nach einem freien Platz gesucht.

Gegeben: Eine normale Hashfunktion:

$$h' : K \to \{0, 1, \ldots, m-1\}$$

Quadratisches Sondieren: Sondierungsfunktion lautet nun:

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m$$

Dabei sind $c_1$ und $c_2$ geeignet gewählte Konstanten.

## Beispiel

Beispiel: $m = 8$, $h'(k) = k \mod m$, $c_1 = c_2 = \frac{1}{2}$, gleiche Schlüssel wie vorhin.

Schlüssel und Wert der Hashfunktion:

$$\begin{array}{c|cccccc} k & 10 & 19 & 31 & 22 & 14 & 16 \\ \hline h'(k) & 2 & 3 & 7 & 6 & 6 & 0 \end{array}$$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| | | 10 | 19 | | | 22 | 31 |

$$\begin{aligned} 14 \to 6 \quad &\to \quad 6 + \frac{1}{2}1 + \frac{1}{2}1^2 \bmod 8 = 7 \\ &\to \quad 6 + \frac{1}{2}2 + \frac{1}{2}2^2 \bmod 8 = 1 \end{aligned}$$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 16 | 14 | 10 | 19 | | | 22 | 31 |

## Beispiel und Analyse

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 16 | 14 | 10 | 19 | | | 22 | 31 |

Durchschnittliche Zeit: Für eine erfolgreiche Suche $\frac{8}{6} \approx 1.33$.

$$\begin{array}{c|ccccccccccccc} k & 10 & & 19 & & 31 & & 22 & & 14 & & 16 & & \\ \hline & 1 & + & 1 & + & 1 & + & 1 & + & 3 & + & 1 & = & 8 \end{array}$$

Probleme: Primäre Häufungen werden vermieden, aber ein anderes Phänomen, die sekundären Häufungen (*secondary clustering*) können auftreten.

---

# Güte von Kollisionsbehandlung

Theoretische Analyseergebnisse:

Durchschnittliche Anzahl der Sondierungen für große $m$, $n$:

| $\alpha$ | Verkettung | | offene Hashverfahren | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | lineares S. | | quadr. S. | | unif. hashing | |
| | erfolgreich | erfolglos | er | el | er | el | er | el |
| 0.5 | 1.250 | 0.50 | 1.5 | 2.5 | 1.44 | 2.19 | 1.39 | 2 |
| 0.9 | 1.450 | 0.90 | 5.5 | 50.5 | 2.85 | 11.40 | 2.56 | 10 |
| 0.95 | 1.475 | 0.95 | 10.5 | 200.5 | 3.52 | 22.05 | 3.15 | 20 |
| 1.0 | 1.500 | 1.00 | — | — | — | — | — | — |

er: erfolgreiche Suche, el: erfolglose Suche

Ergebnisse von D.E. Knuth: The Art of Computer Programming, Vol.3: Sorting and Searching, Addison-Wesley, 1973.

## Double Hashing

Idee: Die Effizienz des uniformen Sondierens wird bereits annähernd erreicht, wenn man statt einer zufälligen Permutation für die Sondierungsfolge eine zweite Hashfunktion verwendet.

Gegeben: Zwei Hashfunktionen:

$$h_1, h_2 : K \to \{0, 1, \ldots, m-1\}$$

Double Hashing: Wir definieren für $i = 0, 1, \ldots, m-1$:

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m$$

Wahl von $h_2(k)$: Für alle Schlüssel $k$ muss die Sondierungsfolge alle Plätze $0, \ldots, m-1$ erreichen. Das bedeutet, dass $h_2(k) \neq 0$ sein muss und $m$ nicht teilen darf. $m$ sollte eine Primzahl sein, $h_2$ sollte unabhängig von $h_1$ gewählt werden.

### Beispiel

$m = 7$, $h_1(k) = k \mod 7$, $h_2(k) = 1 + (k \mod 5)$

| $k$ | 10 | 19 | 31 | 22 | 14 | 16 |
|---|---|---|---|---|---|---|
| $h_1(k)$ | 3 | 5 | 3 | 1 | 0 | 2 |
| $h_2(k)$ | 1 | 5 | 2 | 3 | 5 | 2 |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | | | 10 | | 19 | |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 31 | 22 | | 10 | | 19 | |
| (3) | | | (1) | | (2) | |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 31 | 22 | 16 | 10 | | 19 | 14 |
| (1) | (4) | | (3) | | (2) | (5) |

Durchschnittliche Zeit: Für eine erfolgreiche Suche ist $\frac{12}{6} = 2$. Dies ist jedoch ein untypisch schlechtes Beispiel für Double Hashing.

| $k$ | 10 | | 19 | | 31 | | 22 | | 14 | | 16 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | + | 1 | + | 3 | + | 1 | + | 5 | + | 1 | = | 12 |

### Praxis

- Im Allgemeinen ist Double Hashing effizienter als quadratisches Sondieren.
- In der Praxis entsprechen die Ergebnisse von Double Hashing nahezu denen von uniformen Hashing.

---

## Verbesserung nach Brent [1973]

Idee: Wenn beim Einfügen eines Schlüssels ein sondierter Platz $j$ mit $k' = T[j].key$ belegt ist, setze

$$j_1 = (j + h_2(k)) \mod m$$
$$j_2 = (j + h_2(k')) \mod m.$$

Ist nun $j_1$ besetzt aber $j_2$ frei, verschiebe $k'$ auf $j_2$ um Platz für $k$ auf $j_1$ zu machen.

Angewendet auf unser Beispiel:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | | | 10 | | 19 | |
| | | | 31 | | | |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | | | 31 | 10 | 19 | |

Rest immer frei

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 14 | 22 | 16 | ~~10~~ | 10 | 19 | |
| | | | 31 | | | |

Durchschnittliche Zeit: Für eine erfolgreiche Suche ist $\frac{7}{6} \approx 1.17$.

| $k$ | 10 | | 19 | | 31 | | 22 | | 14 | | 16 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 2 | + | 1 | + | 1 | + | 1 | + | 1 | + | 1 | = | 7 |

### Einfügen nach Brent

Eingabe: Hashtabelle $T$ und neuer Schlüssel $k$.

```
Insert-Brent(T, k):
j ← h1(k)
while T[j].status = used
    k' ←T[j].key
    j1 ←(j + h2(k)) mod m
    j2 ←(j + h2(k')) mod m
    if T[j1].status ≠ used oder T[j2].status = used
        j ← j1
    else
        T[j] ← k
        k ← k'
        j ← j2
T[j] ← k
T[j].status ← used
```

### Analyseergebnis zur Verbesserung nach Brent

Anzahl der Sondierungen: Im Durchschnitt für große $m$ und $n$:

- Erfolgreiche Suche $\approx \frac{1}{1-\alpha}$
- Erfolgreiche Suche $< 2.5$ (unabhängig von $\alpha$ für $\alpha \leq 1$)

Vorteil: Durchschnittlicher Aufwand einer erfolgreichen Suche liegt selbst im Extremfall $\alpha = 1$ in $\Theta(1)$.

---

## Offene Hashverfahren: Eignung und Reorganisation

- Bei offenen Hashverfahren ist der Belegungsfaktor $\alpha = \frac{n}{m}$ immer kleiner (max. gleich) 1, offensichtlich können nicht mehr als $m$ Elemente gespeichert werden.
- Belegungsfaktoren sehr nahe 1 sind i.A. ungünstig, da die Anzahl der zu sondierenden Positionen sehr groß werden kann!
- Gegebenenfalls ist eine Reorganisation notwendig, d.h., dass eine gänzlich neue Hashtabelle z.B. mit doppelter Größe aufgebaut wird, wenn mehr als die ursprünglich erwartete Anzahl an Elementen zu speichern ist (Aufwand i.A. $\Theta(n)$).
- Generell sind offene Hashverfahren besser geeignet, wenn die Anzahl der zu speichernden Elemente vorab bekannt ist und selten oder gar nicht Elemente entfernt werden; häufiges Entfernen bewirkt, dass sich die Sondierungsketten für die Suche verlängern; eine regelmäßige Reorganisation kann dann ebenfalls sinnvoll bzw. notwendig werden.