

Hashing

Algorithmen und Datenstrukturen 1

VU 186.813, 4h, 6 ECTS, SS 2016

Letzte Änderung: 28. April 2016



ALGORITHMS AND
COMPLEXITY GROUP

Hashing

Hashing:

- Alternative Möglichkeit zum Wörterbuchproblem (siehe Kapitel über Suchbäume).
- **Beobachtung:** Im Allgemeinen ist lediglich eine kleine Teilmenge K aller möglichen Schlüssel \mathcal{K} gespeichert.
- **Idee:** Statt in einer Menge von Datensätzen durch Schlüsselvergleiche zu suchen, **ermittle die Position** eines Elements im Speicher (bzw. einem Array) **durch eine arithmetische Berechnung**.

Hashing: Beispiel

Beispiel:

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

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

Hashing: Grundlagen

Hashtabelle:

- Wir gehen davon aus, dass die Hashtabelle T mit einer vorgegebenen Tabellengröße m als Array mit den Indizes $0, \dots, m - 1$ 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 $h : \mathcal{K} \rightarrow \{0, \dots, m - 1\}$, die jedem Schlüssel $k \in \mathcal{K}$ einen eindeutigen – aber i.A. nicht umgekehrt eindeutigen – Hashwert zuordnet.

Hashing

Vorteil: Laufzeit für die Operationen Suchen, Einfügen und Entfernen liegt im Idealfall in $\Theta(1)$, wenn wir davon ausgehen, 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.
- $\Theta(1)$ gilt nur unter der Annahme, dass die Anzahl der Kollisionen vernachlässigbar klein ist.

Hashing

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 $0, \dots, m - 1$ der Hashtabelle aufgeteilt werden.
- Auch kleinste Änderungen im Schlüssel sollen zu einem anderen, möglichst unabhängigen Hashwert führen.

Divisions-Rest-Methode

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

Berechnung:

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

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**.

Divisions-Rest-Methode

Berechnung:

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

Schlechte Wahl für m :

- $m = 2^i$: Nur die letzten i Binärziffern spielen eine Rolle!
- $m = 10^i$: Analog bei Dezimalzahlen.
- $m = r^i$: Analog bei r -adischen Zahlen.
- aber auch $m = r^i \pm j$ für kleines j kann problematisch sein:
z.B.: $m = 2^7 - 1 = 127$: Buchstaben als Zahlen interpretieren
(`'p'` = 112 in ASCII)
 $h(\text{"pt"}) = (112 \cdot 128 + 116) \bmod 127 = 14452 \bmod 127 = 101$
 $h(\text{"tp"}) = (116 \cdot 128 + 112) \bmod 127 = 14960 \bmod 127 = 101$
(Schlüssel in denen zwei Buchstaben vertauscht sind haben hier häufig den gleichen Hashwert)

Divisions-Rest-Methode für Strings

Schlüssel: String $s = (s_1, \dots, s_l) \in \{0, \dots, 127\}^l$

Berechnung: $k = 128^{l-1}s_1 + 128^{l-2}s_2 + \dots + s_l$

Die sehr großen ganzzahligen Werte sind problematisch!

Berechnung mit Horner-Schema:

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

Es gilt:

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

Konsequenz: Keine Zwischenresultate $> (m - 1) \cdot 128 + 127$

Berechnung mit üblichen integer-Typen so gut möglich.

Multiplikationsmethode

Grundlegende Idee:

- Wir nehmen wieder an: Schlüssel $k \in \mathbb{N}$
- **Berechnung:** Annahme, dass $k \in \mathbb{N}$

$$h(k) = \lfloor m \underbrace{(k \cdot A - \lfloor k \cdot A \rfloor)}_{\in [0,1)} \rfloor$$

- Der Schlüssel wird mit einer irrationalen Zahl 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 Schlüsselfolge $1, 2, 3, \dots, i$ liegt der Hashwert des nächsten Schlüssels $i + 1$ immer in einem größten Intervall zwischen allen zuvor ermittelten Hashwerten, 0 und 1.

Multiplikationsmethode: Wahl für A

Allgemein:

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

Beste Wahl für A : Der goldene Schnitt

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

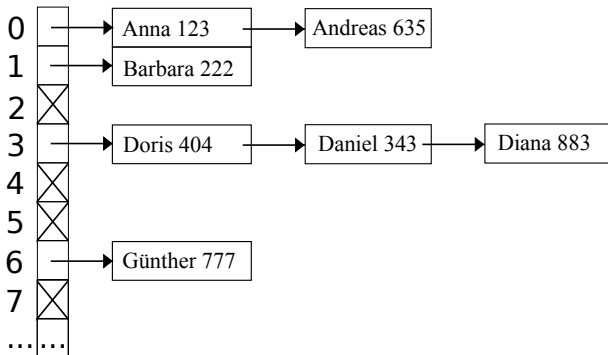
Dabei gilt, dass $\Phi^{-1} = \lim_{n \rightarrow \infty} \frac{F_{n-1}}{F_n}$, wobei F_n die n -te Fibonacci-Zahl ist.

- Eine Begründung dafür findet sich in:
*D.E. Knuth: The Art of Computer Programming, Vol.3:
Sorting and Searching, Addison-Wesley, 1973.*

Kollisionsbehandlung Verkettung der Überläufer

Verkettung der Überläufer

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  $i \leftarrow 0$  bis  $m - 1$   
     $T[i] = null$ 
```

Einfügen

Eingabe: Hashtabelle T und einzufügendes Element p .

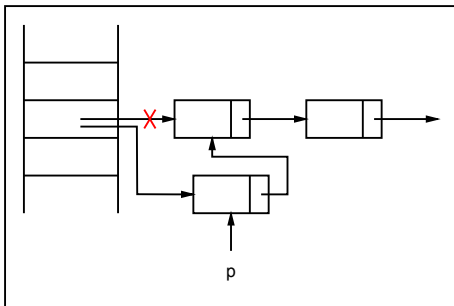
Ergebnis: Hashtabelle T mit neu eingetragenen Element p .

```
Insert( $T, p$ ):  
 $pos \leftarrow h(p.key)$   
 $p.next \leftarrow T[pos]$   
 $T[pos] \leftarrow p$ 
```

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

Einfügen eines Elements

Einfügen von Element p : Einfügen an erster Stelle in der verketteten Liste der Überläufer.



Suchen

Eingabe: Hashtabelle T und gesuchter Schlüssel k .

Rückgabewert: Gesuchtes Element p .

```
Search( $T$ ,  $k$ ):  
   $p = T[h(k)]$   
  while  $p \neq null$  und  $p.key \neq k$   
     $p \leftarrow p.next$   
  return  $p$ 
```

Entfernen eines Elements

Eingabe: Hashtabelle T und Schlüssel k des zu entfernenden Elements (wir gehen davon aus, dass ein Element mit dem gesuchten Schlüssel in T enthalten ist).

Ergebnis: Element mit dem Schlüssel k wurde aus T 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

Offene Hashverfahren

Idee: Alle Elemente werden – im Gegensatz zur Verkettung der Überläufer – direkt im Array gespeichert.

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

Sondierungsreihenfolge (*probing*): Reihenfolge der auszuprobierenden Plätze.

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

Offene Hashverfahren: Sondierung

- Zu jedem Platz $i = 0, \dots, m - 1$ wird ein Flag $f_i \in \{\text{frei}, \text{besetzt}, \text{wiederfrei}\}$ gespeichert.
- In der anfangs leeren Tabelle gilt für alle i $f_i = \text{frei}$.
- Beim Einfügen wird das neue Element am ersten mit *frei* oder *wiederfrei* markierten Platz eingefügt, das Flag wird auf *besetzt* gesetzt.
- Die Suche durchmustert alle Plätze bis der gesuchte Schlüssel entweder gefunden wird oder ein Platz als *frei* markiert ist (Schlüssel nicht enthalten).
- Das Entfernen setzt das Flag für das zu entfernende Element auf *wiederfrei*.

Uniform Hashing

Uniform Hashing:

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

Lineares Sondieren

Gegeben: Eine normale Hashfunktion:

$$h' : \mathcal{K} \rightarrow \{0, 1, \dots, m - 1\}.$$

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

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

Lineares Sondieren: 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

Lineares Sondieren: Probleme

Probleme:

- Im Beispiel ist nach Einfügen von 10 und 19 die Wahrscheinlichkeit für einen neu einzufügenden Schlüssel in der Hashtabelle an einer gewissen Hashadresse gespeichert zu werden für die verschiedenen Hashadressen stark unterschiedlich.
- Im Eintrag $T[4]$ werden nach dem Einfügen von 10 und 19 alle Schlüssel k mit $h(k) = 2$ oder $h(k) = 3$ oder $h(k) = 4$ gespeichert, im Eintrag $T[5]$ dagegen nur alle Schlüssel k mit $h(k) = 5$.
- Bei einer uniformen Hashfunktion, hat in der beschriebenen Situation $T[4]$ die Wahrscheinlichkeit $3/8$ mit dem nächsten Schlüssel belegt zu werden, während diese Wahrscheinlichkeit für $T[5]$ nur $1/8$ beträgt.

Lineares Sondieren: Probleme

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 α dem Wert 1 nähert.

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:

$$h' : \mathcal{K} \rightarrow \{0, 1, \dots, m - 1\}.$$

Quadratisches Sondieren: Sondierungsfunktion lautet nun:

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

Dabei sind c_1 und c_2 geeignet gewählte Konstanten.

Quadratisches Sondieren: Beispiel

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

Schlüssel und Wert der Hashfunktion:

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

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

$$\begin{aligned} 14 \rightarrow 6 &\rightarrow 6 + \frac{1}{2}(1 + 1^2) \bmod 8 = 7 \\ &\rightarrow 6 + \frac{1}{2}(2 + 2^2) \bmod 8 = 1 \end{aligned}$$

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

Quadratisches Sondieren: Beispiel und Analyse

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

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

k	10		19		31		22		14		16	
	1	+	1	+	1	+	1	+	3	+	1	= 8

Probleme: Es wird zwar die primäre Häufung vermieden, aber ein anderes Phänomen, die sekundäre Häufung (*secondary clustering*) kann jetzt auftreten.

Güte von Kollisionsstrategien

Theoretische Analyseergebnisse:

Durchschnittliche Anzahl der Sondierungen (für große m , n):

α	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:

$$h_1, h_2 : \mathcal{K} \rightarrow \{0, 1, \dots, m-1\}.$$

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

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m.$$

Wahl von $h_2(k)$: Für alle Schlüssel k muss die Sondierungsfolge alle Plätze $0, \dots, m$ 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.

Double Hashing

Beispiel: $m = 7$, $h_1(k) = k \bmod 7$, $h_2(k) = 1 + (k \bmod 5)$

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

Verbesserung nach Brent [1973]

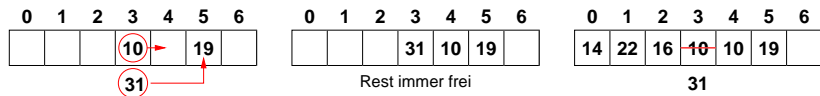
Idee: Wenn beim Einfügen eines Schlüssels ein sondierter Platz j ist mit $k' = T[j].key$ belegt ist, setze

$$j_1 = (j + h_2(k)) \bmod m$$

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

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

Angewendet auf unser Beispiel:



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( $T, k$ ):
```

```
 $j \leftarrow h_1(k)$ 
```

```
while  $T[j].status = used$ 
```

```
     $k' \leftarrow T[j].key$ 
```

```
     $j_1 \leftarrow (j + h_2(k)) \bmod m$ 
```

```
     $j_2 \leftarrow (j + h_2(k')) \bmod m$ 
```

```
    if  $T[j_1].status \neq used$  oder  $T[j_2].status = used$ 
```

```
         $j \leftarrow j_1$ 
```

```
    else
```

```
         $T[j] \leftarrow k$ 
```

```
         $k \leftarrow k'$ 
```

```
         $j \leftarrow j_2$ 
```

```
 $T[j] \leftarrow k$ 
```

```
 $T[j].status \leftarrow used$ 
```

Analyseergebnis zur Verbesserung nach Brent

Anzahl der Sondierungen: Im Durchschnitt (für große m und n):

- Erfolglose Suche $\approx \frac{1}{1-\alpha}$
- Erfolgreiche Suche < 2.5 (unabhängig von α 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.