Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement Informatik
Markus Püschel
Peter Widmayer
Thomas Tschager
Tobias Pröger

24. November 2016

Algorithmen & Datenstrukturen Lösungen zu Blatt 9 HS 16

Lösung 9.1 *Offene Hashverfahren.*

Die im folgenden benutzten Sondierungsverfahren werden in genau der Form benutzt, in der sie in der Vorlesung vorgestellt wurden, d.h. die Sondierungsfunktion wird von der Hashfunktion subtrahiert. Auch eine Addition der Funktionen ist gültig, führt aber u.U. zu anderen Lösungen.

a)
- $h(k)$ = Quersumme von k. Diese Funktion ist ungeeignet für Hashing, da der Wert der Hashfunktion zwischen 0 und $p-1$ liegen muss, die Quersumme aber beliebig gross werden kann. Selbst wenn die Quersumme stets kleiner als p wäre, wäre sie ungeeignet, da sie schlecht streut.
- $h(k) = k(1+p^3) \bmod p$. Wegen $p^3 \bmod p = 0$ ist $h(k) = k \bmod p$, was wie in der Vorlesung erklärt eine geeignete Hashfunktion ist. Der einzige Nachteil der Formulierung wie in der Aufgabenstellung sind die überflüssigen arithmetischen Operationen.
- $h(k) = \lfloor p(rk - \lfloor rk \rfloor) \rfloor$, $r \in \mathbb{R}^+ \backslash \mathbb{Q}$. Diese Hashfunktion entspricht der in der Vorlesung vorgestellten *multiplikativen Methode* und ist gut, wenn r geeignet gewählt wird. Man kann zeigen, dass für $r = \phi^{-1} = \frac{\sqrt{5}-1}{2}$ besonders gute Ergebnisse erzielt werden.

b) (i) 17: $h(17) = 6$

						17				

6: $h(6) = 6 \to h(6) - 1 \equiv 5$

					6	17				

5: $h(5) = 5 \to h(5) - 1 \equiv 4$

				5	6	17				

8: $h(8) = 8$

				5	6	17		8		

11: $h(11) = 0$

11				5	6	17		8		

28: $h(28) = 6 \to h(28) - 1 \equiv 5 \to h(28) - 2 \equiv 4 \to h(28) - 3 \equiv 3$

11			28	5	6	17		8		

14: $h(14) = 3 \to h(14) - 1 \equiv 2$

11		14	28	5	6	17		8		

15: $h(15) = 4 \to h(15) - 1 \equiv 3 \to h(15) - 2 \equiv 2 \to h(15) - 3 \equiv 1$

11	15	14	28	5	6	17		8		

Kollisionen: 9

(ii) 17: $h(17) = 6$

						17				

6: $h(6) = 6 \to h(6) - 1 \equiv 5$

					6	17				

5: $h(5) = 5 \to h(5) - 1 \equiv 4$

				5	6	17				

8: $h(8) = 8$

				5	6	17		8		

11: $h(11) = 0$

11				5	6	17		8		

28: $h(28) = 6 \to h(28) - 1 \equiv 5 \to h(28) + 1 \equiv 7$

11				5	6	17	28	8		

14: $h(14) = 3$

11			14	5	6	17	28	8		

15: $h(15) = 4 \to h(15) - 1 \equiv 3 \to h(15) + 1 \equiv 5 \to h(15) - 4 \equiv 0$
$\to h(15) + 4 \equiv 8 \to h(15) - 9 \equiv 6 \to h(15) + 9 \equiv 2$

11		15	14	5	6	17	28	8		

Kollisionen: 10

(iii) 17: $h(17) = 6$

						17				

6: $h(6) = 6 \to h(6) - h'(6) \equiv 10$

						17				6

5: $h(5) = 5$

					5	17				6

8: $h(8) = 8$

					5	17		8		6

11: $h(11) = 0$

11					5	17		8		6

28: $h(28) = 6 \to h(28) - h'(28) \equiv 4$

11				28	5	17		8		6

14: $h(14) = 3$

11			14	28	5	17		8		6

15: $h(15) = 4 \to h(15) - h'(15) \equiv 8 \to h(15) - 2h'(15) \equiv 1$

11	15		14	28	5	17		8		6

Kollisionen: 4

c) Die Löschung eines Schlüssels k ist problematisch, wenn ein weiterer Schlüssel k' mit der gleichen Hashadresse später als k eingefügt wurde. Würde k einfach gelöscht (z.B. indem die Position als frei markiert wird), dann könnte der Schlüssel k' nicht mehr gefunden werden, da das Sondieren endet, sobald eine freie Position gefunden wird. Im gegebenen Beispiel könnten die Schlüssel 6 und 28 nicht mehr gefunden werden. Folglich muss die Position explizit als gelöscht markiert werden, und bei der Suche nach einem Schlüssel muss die Sondierung fortgesetzt werden, sobald eine solche Position gefunden wird. Selbstverständlich darf die Markierung beim späteren Einfügen eines anderen Schlüssels überschrieben werden, sofern dies nötig ist.

Werden nun viele Schlüssel gelöscht, dann kann es passieren, dass die Suche nach einem Schlüssel sehr ineffizient wird (denn die Sondierung besucht u.U. viele als gelöscht markierte Positionen). Hashing ist also besonders dann geeignet, wenn Schlüssel grösstenteils nur eingefügt und gesucht und nur selten gelöscht werden.

d)
- $h'(k) = \lceil \ln(k+1) \rceil \bmod q$. Diese Funktion ist ungeeignet als zweite Hashfunktion, denn für den Schlüssel $k = 0$ ist $h'(0) = \lceil \ln(1) \rceil = 0$.
- $s(j,k) = k^j \bmod p$. Diese Funktion ist als Sondierungsfunktion ungeeignet, denn für die Schlüssel $k = 0$ oder $k = 1$ hat $s(j,k)$ konstant den Wert 0 bzw. 1.
- $s(j,k) = ((k \cdot j) \bmod q) + 1$. Diese Funktion ist ebenfalls als Sondierungsfunktion ungeeignet, denn ihr Wert ist konstant 1, wenn der Schlüssel k ein Vielfaches von q ist. Zudem ist für alle anderen Schlüssel die Bildmenge von $s(j,k)$ genau $\{1, \ldots, q\}$, d.h. $p - q$ Adressen der Hashtabelle können überhaupt nicht erreicht werden.

e) Beim quadratisches Sondieren hängt die Folge der $s(j,k)$ nicht vom Schlüssel k ab. Gibt es nun viele Schlüssel k, die auf die gleiche Hashadresse $h(k)$ abgebildet werden, dann ist die Sondierungsfolge für all diese Schlüssel gleich. Also treten beim Sondieren viele Kollisionen auf. Man spricht dann auch von *sekundärem Clustering*. Double Hashing vermeidet sekundäres Clustering, da verschiedene Schlüssel k, k' mit $h(k) = h(k')$ oftmals unterschiedliche Sondierungsfolgen besitzen.

Lösung 9.2 *Kuckucks-Hashing (Cuckoo Hashing).*

a)
- Einfügen von 27:

T_1:			27			T_2:					

- Einfügen von 2 (verdrängt 27 in T_1):

T_1:			2			T_2:	27				

- Einfügen von 32 (verdrängt 2 in T_1, 2 verdrängt 27 in T_2, 27 verdrängt 32 in T_1):

T_1:			27			T_2:	2	32			

b) Hier sind verschiedene Wahlen möglich. So zum Beispiel führt die Einfügung des Schlüssels 7 zu einer Endlosschleife.

1) Initial:

T_1:			27			T_2:	2	32			

2) Füge 7 in T_1 ein (verdrängt 27):

T_1:			7			T_2:	2	32			

3) Füge 27 in T_2 ein (verdrängt 2):

T_1:			7				T_2:	27	32			

4) Füge 2 in T_1 ein (verdrängt 7):

T_1:			2				T_2:	27	32			

5) Füge 7 in T_2 ein (verdrängt 32):

T_1:			2				T_2:	27	7			

6) Füge 32 in T_1 ein (verdrängt 2):

T_1:			32				T_2:	27	7			

7) Füge 2 in T_2 ein (verdrängt 27):

T_1:			32				T_2:	2	7			

8) Füge 27 in T_1 ein (verdrängt 32):

T_1:			27				T_2:	2	7			

9) Füge 32 in T_2 ein (verdrängt 7):

T_1:			27				T_2:	2	32			

Die Situation jetzt ist identisch zur Ausgangssituation, in der der Schlüssel 7 in T_1 eingefügt werden soll.

Lösung 9.3 *Queue mit Stacks implementieren.*

Zur Erinnerung: In der amortisierten Analyse mittels Potentialfunktion definiert man Φ_i als das Potential (d.h. dem Bankkontostand) nach der i-ten Operation. Die i-te Operation habe tatsächliche Kosten t_i. Dann sind die *amortisierten Kosten* der i-ten Operation definiert als $a_i := t_i + \Phi_i - \Phi_{i-1}$. Mit dieser Definition folgt für eine Folge von m Operationen

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} (t_i + \Phi_i - \Phi_{i-1}) = \left(\sum_{i=1}^{m} t_i \right) + \Phi_m - \Phi_0, \tag{1}$$

und somit erhalten wir

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} a_i + \Phi_0 - \Phi_m. \tag{2}$$

Wenn es also gelingt, die amortisierten Kosten jeder Operation sowie den Term $\Phi_0 - \Phi_m$ abzuschätzen, dann erhält man so auch eine Abschätzung für die tatsächlichen Gesamtkosten. Wird die Potenzialfunktion beispielsweise so gewählt, dass $\Phi_m \geq \Phi_0$ für jedes m, dann folgt $\sum_{i=1}^{m} t_i \leq \sum_{i=1}^{m} a_i$, d.h. die tatsächlichen Gesamtkosten können durch die Summe der amortisierten Kosten nach oben abgeschätzt werden.

Wir bezeichnen die beiden Stacks als Stack_L und Stack_R. Diese seien initial leer. Eine ENQUEUE-Operation legt das gegebene Objekt x auf Stack_L ab (mit der Operation PUSH). Bei einer DEQUEUE-Operation wird das oberste Objekt von Stack_R ausgegeben und entfernt (mit POP), falls Stack_R nicht leer ist. Ansonsten entfernen wir jeweils das oberste Element von Stack_L (mit POP) und fügen es Stack_R (mittels PUSH) hinzu. Dieses Vorgehen wiederholen wir, bis Stack_L leer ist. Anschliessend wird das oberste Element von Stack_R ausgegeben und von Stack_R entfernt.

Wir definieren nun die Potentialfunktion Φ_i als

$$2 \cdot \text{Anzahl der momentan gespeicherten Objekte auf Stack}_L. \tag{3}$$

Eine ENQUEUE-Operation hat tatsächliche Kosten $t_i = 1$ und amortisierte Kosten

$$a_i = t_i + \Phi_i - \Phi_{i-1} = 1 + 2 = 3. \tag{4}$$

Bei einer DEQUEUE-Operation gehen wir zunächst davon aus, dass Stack_R nicht leer ist. Dann sind die tatsächlichen und amortisierten Kosten $t_i = a_i = 1$. Nun sei Stack_R leer und k die Anzahl der Elemente auf Stack_L. Die tatsächlichen Kosten sind dann $t_i = 2k + 1$. Die amortisierten Kosten sind

$$a_i = t_i + \Phi_i - \Phi_{i-1} = 2k + 1 - 2k = 1. \tag{5}$$

Damit sind die Operationen ENQUEUE und DEQUEUE in amortisierter Zeit $\Theta(1)$ ausführbar.