Prof. Dr. Ir. Joost-Pieter Katoen

Christian Dehnert, Friedrich Gretz, Benjamin Kaminski, Thomas Ströder

Tutoraufgabe 1 (Güte von Hashfunktionen):

Untersuchen Sie, inwiefern sich die folgenden Funktionen für die Verwendung als Hashfunktion eignen. Begründen Sie Ihre Antwort.

- $f: \{0, 1, 2, \ldots, 100\} \to \{0, 1, 2, \ldots, 10\},\ x \mapsto \left\lfloor \frac{x}{10} \right\rfloor$
- $g: \mathbb{N} \to \mathbb{Z}/100\mathbb{Z},\ x \mapsto 2^x \bmod 100$
- $h: \{0, 1, 2, \ldots, 100\} \to \{0, 1, 2, \ldots, 10\},\ x \mapsto x \bmod 10$
- $i: \mathbb{N} \to \{0, 1, 2, \ldots, 50\},\ x \mapsto \bmod 51$

Lösung:

Die Funktion f ist *einfach zu berechnen*, *surjektiv* und verteilt die Werte ihres Definitionsbereiches *mit gleicher Häufigkeit* auf ihren Bildbereich. Allerdings werden *ähnliche Schlüssel nicht* möglichst breit auf den Bildbereich *verteilt*, da bspw. alle Werte von 0 bis 9 auf 0 abgebildet werden, alle Werte von 10 bis 19 auf 1, usw.

Die Funktion g ist *einfach zu berechnen* aber *nicht surjektiv*: Mit Ausnahme der 1 werden keine ungeraden Zahlen getroffen. Auf die geraden Zahlen und die 1 verteilt g die Werte ihres Definitionsbereiches *annähernd mit gleicher Häufigkeit*; nur die 1 und die 2 werden lediglich ein Mal getroffen. Auch werden *ähnliche Schlüssel* sehr breit auf den Bildbereich *verteilt*.

Die Funktion h ist *einfach zu berechnen* aber *nicht surjektiv*, da die 10 nie getroffen wird. Auf die Zahlen von 0 bis 9 verteilt h die Werte ihres Definitionsbereiches *mit gleicher Häufigkeit*. Auch werden *ähnliche Schlüssel* einigermaßen breit auf den Bildbereich *verteilt*. Die Streuung ist jedoch nicht so gut wie die von g.

Die Funktion i erfüllt alle Kriterien einer guten Hashfunktion. Jedoch ist auch ihre Streuung nicht so gut wie die von g.

Tutoraufgabe 2 (Countingsort):

a) Das folgendes Array ist mit Countingsort zu sortieren. Geben Sie das Histogramm- und das Positionsarray vor dem ersten Einfügen ins Ausgabearray an, sowie das Positions- und Ausgabearray nach jedem Einfügeschritt.

4	3	0	1	4	2	3	7	3

b) Der in der Vorlesung vorgestellte Algorithmus Countingsort fügt die Elemente des Eingabearrays von hinten nach vorne in das Ausgabearray ein. Welche Nachteile ergäben sich, wenn man das Eingabearray stattdessen von vorne nach hinten durchlaufen würde?

Lösung:

a) Eingabearray

4	3	0	1	4	2	3	7	3

Histogramm

1	1	1	3	2	0	0	1
0	1	2	3	4	5	6	7

Positionen

1	2	3	6	8	8	8	9
0	1	2	3	4	5	6	7

Ausgabearray

0	1	2	3	4	5	6	7	8

Eingabearray

4	3	0	1	4	2	3	7	3

Positionen

1	2	3	5	8	8	8	9
0	1	2	3	4	5	6	7

Ausgabearray

					3			
0	1	2	3	4	5	6	7	8

Eingabearray

4	3	0	1	4	2	3	7	3

Positionen

1	2	3	5	8	8	8	8
0	1	2	3	4	5	6	7

Ausgabearray

					3			7
0	1	2	3	4	5	6	7	8

Eingabearray

4	3	0	1	4	2	3	7	3

Positionen

1	2	3	4	8	8	8	8
0	1	2	3	4	5	6	7

Ausgabearray

				3	3			7
0	1	2	3	4	5	6	7	8

Eingabearray

4	3	0	1	4	2	3	7	3

Positionen

1	2	2	4	8	8	8	8
0	1	2	3	4	5	6	7

Ausgabearray

		2		3	3			7
0	1	2	3	4	5	6	7	8

Eingabearray

4	3	0	1	4	2	3	7	3

Positionen

1	2	2	4	7	8	8	8
0	1	2	3	4	5	6	7

Ausgabearray

		2		3	3		4	7
0	1	2	3	4	5	6	7	8

Eingabearray	**4**	**3**	**0**	**1**	**4**	**2**	**3**	**7**	**3**

Positionen	**1**	**1**	**2**	**4**	**7**	**8**	**8**	**8**
	0	1	2	3	4	5	6	7

Ausgabearray		**1**	**2**		**3**	**3**		**4**	**7**
	0	1	2	3	4	5	6	7	8

Eingabearray	**4**	**3**	**0**	**1**	**4**	**2**	**3**	**7**	**3**

Positionen	**0**	**1**	**2**	**4**	**7**	**8**	**8**	**8**
	0	1	2	3	4	5	6	7

Ausgabearray	**0**	**1**	**2**		**3**	**3**		**4**	**7**
	0	1	2	3	4	5	6	7	8

Eingabearray	**4**	**3**	**0**	**1**	**4**	**2**	**3**	**7**	**3**

Positionen	**0**	**1**	**2**	**3**	**7**	**8**	**8**	**8**
	0	1	2	3	4	5	6	7

Ausgabearray	**0**	**1**	**2**	**3**	**3**	**3**		**4**	**7**
	0	1	2	3	4	5	6	7	8

Eingabearray	**4**	**3**	**0**	**1**	**4**	**2**	**3**	**7**	**3**

Positionen	**0**	**1**	**2**	**3**	**6**	**8**	**8**	**8**
	0	1	2	3	4	5	6	7

Ausgabearray	**0**	**1**	**2**	**3**	**3**	**3**	**4**	**4**	**7**
	0	1	2	3	4	5	6	7	8

b) Fängt man beim ersten Element an, dann ist der Algorithmus nicht mehr stabil.

Tutoraufgabe 3 (Hashing):

a) Gegeben sei eine Hash-Table der Größe m und eine beliebige Hash-Funktion $h\colon U \to \{0, \ldots, m-1\}$. Die Menge U habe nun mindestens $n \cdot m$ Elemente, also $|U| \geq n \cdot m$. Zeigen Sie, dass U eine Teilmenge U_0 der Größe n besitzt ($|U_0| = n$), so dass

$$h(x_1) = h(x_2) \quad \text{für alle} \quad x_1, x_2 \in U_0$$

Was haben Sie damit für die Worst-Case-Laufzeit der Suche mittels Hashing mit Verkettung bewiesen?

b) Gegeben sei nun eine Hash-Table mit einer initialen Größe von 1000 und eine Hash-Funktion, die ein gleichverteiltes Hashing gewährleistet. Nach wie vielen Einfügungen müssen sie a-priori mit einer Kollisionswahrscheinlichkeit von mehr als 80% rechnen?

Um die Anzahl von Kollisionen beim Hashing gering zu halten, kann man die Größe der Hash-Table nach einer gewissen Anzahl von Einfügungen erhöhen. Nach welcher Anzahl k von Einfügeoperationen muss die Tabelle das erste Mal vergrößert werden, wenn bei den vorhergehenden $k-1$ Einfügeoperationen keine Kollision auftrat und die Wahrscheinlichkeit für eine Kollision bei der k-ten Einfügung weniger als 20% betragen soll? Begründen Sie ihre Antwort.

Lösung:

a) **Beweis durch Widerspruch.** Seien U, h, n und m wie in der Aufgabenstellung gegeben und benenne U^i für $0 \leq i < m$ die Menge der Elemente, die durch h auf i abgebildet werden, also $U^i = h^{-1}(i)$, wobei $h^{-1}(i)$ das Urbild von i unter h ist (und nicht etwa die Umkehrfunktion).

Annahme: es existiert kein $U_0 \subseteq U$ mit $|U_0| = n$, so dass

$$h(x_1) = h(x_2) \quad \text{für alle} \quad x_1, x_2 \in U_0$$

Dann existiert insbesondere für kein i eine Menge $U_1 \subseteq U^i$ mit $|U_1| \geq n$, da sonst jede n-elementige Teilmenge von U_1 der Annahme widerspricht (da für alle $x \in U^i$ der Hash-Wert gleich ist, nämlich i).

Dann folgt:

$$\begin{aligned} & |U^i| < n \text{ für alle } i \in \{0, \ldots, m-1\} \\ \Longrightarrow \quad & |U| = \sum_{i \in \{0,\ldots,m-1\}} |U^i| < n \cdot m \end{aligned}$$

□

Damit ist gezeigt, dass die Worst-Case-Laufzeit der Suche bei Hashing mit Verkettung – unabhängig von der Hashfunktion – in $\Omega(n)$ liegt, wenn U hinreichend groß ist.

b) Nach Vorlesung ist die a-priori Wahrscheinlichkeit für eine Kollision nach k Einfügungen

$$1 - \prod_{i=0}^{k-1} \frac{m-i}{m} \overset{(!)}{\geq} 0.8$$

was äquivalent ist zu

$$\prod_{i=0}^{k-1} \frac{m-i}{m} \overset{(!)}{<} 0.2$$

Durch einfaches Ausprobieren ergibt sich, dass dies für $m = 1000$ ab $k = 57$ der Fall ist.

Vergrößert man die Hash-Table dynamisch nach einigen Einfügungen, so kann man die Information über die bisherigen Einfügungen berücksichtigen. Gab es bei den vorigen $k-1$ Einfügungen keine Kollision, so sind genau $k-1$ Indizes belegt und die Wahrscheinlichkeit für eine Kollision beim nächsten Schritt damit

$$\frac{k-1}{m} \overset{(!)}{<} 0.2$$

Für $m = 1000$ ergibt sich hier $k < 201$. Die Hash-Table muss unter den gegebenen Bedingungen also vor der 201. Einfügung vergrößert werden.

Tutoraufgabe 4 (Hashing):

Gegeben sei eine Hash-Table der Größe 23 und die folgenden beiden Hash-Funktionen über der Universalmenge $U = \{0, \ldots, 499\}$:

- $h_1(x) =$ Quersumme von x
- $h_2(x) = x \mod 23$

a) Fügen Sie die Werte $12, 99, 111, 76, 23, 30$ sowohl mit h_1 als auch h_2 mittels

(i) Hashing mit Verkettung

(ii) Hashing mit linearem Sondieren

in jeweils eine Tabelle ein (es sind also 4 Tabellen zu erstellen). Geben Sie die nicht-leeren Teile der Tabellen nach jedem Einfügeschritt an.

b) Löschen Sie nacheinander die Werte 111, 12 und 76 aus allen Tabellen aus Aufgabe **a)**. Geben Sie die nicht-leeren Teile der Tabellen nach jedem Löschschritt an. Erläutern Sie, welches Vorgehen dafür jeweils nötig ist.

c) Suchen Sie in den Tabellen, die aus Teilaufgabe **b)** resultierten, nach dem Wert 30. Erläutern Sie, welches Vorgehen dafür jeweils nötig ist.

Lösung:

a) (i) 12 einfügen:

h_1	
3	12

h_2	
12	12

99 einfügen:

3	12
18	99

7	99
12	12

111 einfügen:

3	111 → 12
18	99

7	99
12	12
19	111

76 einfügen:

3	111 → 12
13	76
18	99

7	76 → 99
12	12
19	111

23 einfügen:

3	111 → 12
5	23
13	76
18	99

0	23
7	76 → 99
12	12
19	111

30 einfügen:

3	30 → 111 → 12	0	23
5	23	7	30 → 76 → 99
13	76	12	12
18	99	19	111

(ii) 12 einfügen:

h_1		h_2	
3	12	12	12

99 einfügen:

3	12	7	99
18	99	12	12

111 einfügen:

3	12	7	99
4	111	12	12
18	99	19	111

76 einfügen:

3	12	7	99
4	111	8	76
13	76	12	12
18	99	19	111

23 einfügen:

3	12
4	111
5	23
13	76
18	99

0	23
7	99
8	76
12	12
19	111

30 einfügen:

3	12
4	111
5	23
6	30
13	76
18	99

0	23
7	99
8	76
9	30
12	12
19	111

b) Beim Hashing mit Verkettung muss das Element in der jeweiligen Liste gesucht, dann gelöscht und anschließend noch gegebenenfalls Zeiger berichtigt werden.

Beim Hashing mit linearem Sondieren wird jeweils zuerst in dem "Fach" gesucht, das mit dem Hashwert assoziiert ist. Enthält dieses das gesuchte Element, so kann es durch **Deleted** ersetzt werden. Enthält es jedoch nicht das Element sondern **Deleted** oder einen anderen Wert, so muss die Suche mit linearem Sondieren fortgesetzt werden.

111 löschen:

(i)

h_1	
3	30 → 12
5	23
13	76
18	99

h_2	
0	23
7	30 → 76 → 99
12	12

(ii)

3	12
4	**Deleted**
5	23
6	30
13	76
18	99

0	23
7	99
8	76
9	30
12	12
19	**Deleted**

12 löschen:

(i)

h_1	
3	30
5	23
13	76
18	99

h_2	
0	23
7	30 → 76 → 99

(ii)

3	**Deleted**
4	**Deleted**
5	23
6	30
13	76
18	99

0	23
7	99
8	76
9	30
12	**Deleted**
19	**Deleted**

76 löschen:

(i)

h_1

3	30
5	23
18	99

h_2

0	23
7	30 → 99

(ii)

3	**Deleted**
4	**Deleted**
5	23
6	30
13	**Deleted**
18	99

0	23
7	99
8	**Deleted**
9	30
12	**Deleted**
19	**Deleted**

c) Beim Hashing mit Verkettung muss die Liste der Werte, die mit dem Hashwert von 30 assoziiert ist, durchlaufen werden. In diesem Beispiel wird die 30 in beiden Fällen sofort gefunden, da sie einmal das einzige Element der Liste und im anderen Fall das erste Element ist.

Beim Hashing mit linearem Sondieren wird jeweils zuerst in dem "Fach" gesucht, das mit dem Hashwert 30 assoziiert ist. In diesem Beispiel enthalten jedoch beide nicht das Element 30 sondern **Deleted**. Daher muss die Suche mit linearem Sondieren fortgesetzt werden. Dies gilt auch für den Fall, dass an einem Element nicht **Deleted** sondern ein anderer Eintrag steht. Die 30 wird dann nach 3 (für h_1) bzw. 2 (für h_2) Sondierungsschritten gefunden.