Albert-Ludwigs-Universität
Institut für Informatik
Prof. Dr. F. Kuhn
M. Fuchs, G. Schmid

Algorithmen und Datenstrukturen
Sommersemester 2024
Musterlösung Übungsblatt 4

Abgabe: Dienstag, 14. Mai, 2024, 10:00 Uhr

Aufgabe 1: Hashing mit offener Adressierung *(5 Punkte)*

Sei $\mathcal{H}$ eine Hashtabelle der Größe $m = 13$ und seinen $h_1, h_2, h_3 : \mathbb{N}_0 \mapsto \{0, ..., m-1\}$ Hashfunktionen definiert wie folgt[1]:

- $h_1(x) := \overline{x} \mod m$
- $h_2(x) := 3 \cdot x \mod m$
- $h_3(x) := x + 1 \mod m$

Fügen Sie die Schlüssel $23, 12, 75, 945, 30, 99, 345$ (in dieser Reihenfolge) in die initial leere Hashtabelle $\mathcal{H}$ ein. Lösen Sie Konflikte wie folgt:

a) Lineares Sondieren unter der Benutzung von h_1. *(2 Punkte)*

b) Doppel-Hashing unter Benutzung von h_2 und h_3.[2] *(3 Punkte)*

Geben Sie den Zustand der Hashtabelle in jedem Schritt an!

Musterlösung

a)

$$h(x, i) := h_1(x) + i \mod m$$

0	1	2	3	4	5	6	7	8	9	10	11	12
-	-	-	-	-	23	-	-	-	-	-	-	-

0	1	2	3	4	5	6	7	8	9	10	11	12
-	-	-	12	-	23	-	-	-	-	-	-	-

0	1	2	3	4	5	6	7	8	9	10	11	12
-	-	-	12	-	23	-	-	-	-	-	-	75

0	1	2	3	4	5	6	7	8	9	10	11	12
-	-	-	12	-	23	945	-	-	-	-	-	75

0	1	2	3	4	5	6	7	8	9	10	11	12
-	-	-	12	30	23	945	-	-	-	-	-	75

0	1	2	3	4	5	6	7	8	9	10	11	12
-	-	-	12	30	23	945	99	-	-	-	-	75

0	1	2	3	4	5	6	7	8	9	10	11	12
345	-	-	12	30	23	945	99	-	-	-	-	75

[1] Wir definieren $\overline{x}$ als die Quersumme von x.
[2] Es soll also $(h_2(x) + i \cdot h_3(x)) \mod m$ als Hashfunktion verwendet werden.

b)

$$h(x,i) := h_2(x) + i \cdot h_3(x) \mod m$$

0	1	2	3	4	5	6	7	8	9	10	11	12
-	-	-	-	23	-	-	-	-	-	-	-	-

0	1	2	3	4	5	6	7	8	9	10	11	12
-	-	-	-	23	-	-	-	-	-	12	-	-

0	1	2	3	4	5	6	7	8	9	10	11	12
-	-	75	-	23	-	-	-	-	-	12	-	-

0	1	2	3	4	5	6	7	8	9	10	11	12
-	945	75	-	23	-	-	-	-	-	12	-	-

0	1	2	3	4	5	6	7	8	9	10	11	12
-	945	75	-	23	-	-	-	-	-	12	-	30

0	1	2	3	4	5	6	7	8	9	10	11	12
-	945	75	-	23	-	-	-	-	-	12	99	30

0	1	2	3	4	5	6	7	8	9	10	11	12
-	945	75	-	23	-	-	-	345	-	12	99	30

Aufgabe 2: Hashing mit Chaining *(5 Punkte)*

Gegeben sei eine Hash-Table der Größe $m > 0$ und eine beliebige Hashfunktion $h : S \mapsto \{0, ..., m-1\}$. Die Menge S habe mindestens $y \cdot m$ Elemente für ein $y > 0$ (sprich, $|S| \geq y \cdot m$).

a) Zeigen Sie, dass S mindestens eine Teilmenge Y, bestehend aus mindestens y Elementen (also $|Y| \geq y$), besitzt, so dass $h(x_1) = h(x_2)$ für alle $x_1, x_2 \in Y$. *(4 Punkte)*

b) Was sagt uns das Resultat in *a)* über die Worst-Case Laufzeit von *"find"* in einer Hashtabelle mit Chaining aus (wenn unsere Hashtabelle genau die Elemente aus S speichert bevor *"find"* aufgerufen wird)[3]? *(1 Punkt)*

Musterlösung

a) Beweis durch Widerspruch. Wir nehmen also an es existiert keine solche Menge Y der Größe mindestens y. Weiterhin, definieren wir die Mengen $S_0, S_2, ..., S_{m-1}$ wie folgt:

$$S_i := \{x \in S \mid h(x) = i\}$$

Da die Hashfunktion h alle Elemente auf genau einen Wert von 0 bis $m-1$ abbildet, gilt sofort dass

$$|S_0| + |S_1| + \ldots + |S_{m-1}| = |S|. \tag{1}$$

Durch unsere obige Annahme besitzt jede dieser Teilmengen weniger als y Elemente, sprich $|S_i| < y$. Dann würde aber folgende Ungleichung gelten:

$$|S_0| + |S_1| + \ldots + |S_{m-1}| < y + y + \ldots + y = m \cdot y \tag{2}$$

Verknüpfen wir nun (1) und (2) gilt $|S| < m \cdot y$. Widerspruch.

b) Die Suche eines Elements in dieser Datenstruktur benötigt - unabhängig von der Hashfunktion - $\Omega(y)$ Zeit.

[3] Hier geht es darum eine von der Hashfunktion h unahängige worst-case Laufzeit für *find* anzugeben.

Aufgabe 3: Anwendung von Hashtabellen *(10 Punkte)*

Gegeben ist folgender Algorithmus:

Algorithm 1 `algorithm` ▷ Input: Array A of length n with integer entries

```
1: for i = 1 to n − 1 do
2:     for j = 0 to i − 1 do
3:         for k = 0 to n − 1 do
4:             if |A[i] − A[j]| = A[k] then
5:                 return true
6: return false
```

(a) Beschreiben Sie, was `algorithm` berechnet und analysieren Sie die asymptotische Laufzeit. *(3 Punkte)*
Hinweis: Die Differenz $|A[i] - A[j]|$ kann beliebig große Werte annehmen.

(b) Beschreiben Sie einen auf *hashing* basierenden alternativen Algorithmus $\mathcal{B}$ für dieses Problem (d.h. $\mathcal{B}(A) =$ `algorithm`(A) für jede Eingabe A) mit einer Laufzeit von $\mathcal{O}(n^2)$ (mit Begründung). *(3 Punkte)*
Hinweis: Sie dürfen annehmen, dass das Einfügen und Finden von Schlüsseln in einer Hashtabelle $\mathcal{O}(1)$ Zeitschritte benötigt, wenn $\alpha = \mathcal{O}(1)$ (α ist der Load der Hashtabelle).

(c) Beschreiben Sie einen weiteren Algorithmus für dieses Problem ohne Verwendung von Hashing mit einer Laufzeit von $\mathcal{O}(n^2 \log n)$ (mit Begründung). *(4 Punkte)*

Musterlösung

(a) Der Algorithmus testet, ob es zwei Elemente im Array gibt, deren Abstand (Betrag der Differenz) einem Wert im Array entspricht. Falls ja, so gibt der Algorithmus "true" aus, andernfalls "false". In letzterem Fall werden alle Schleifen vollständig durchlaufen. Man betrachtet dabei

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \mathcal{O}(n^2)$$

viele Paare (i, j) und für jedes dieser Paare checkt man n mal die Bedingung in Zeile 4. Die Laufzeit ist also $\mathcal{O}(n^3)$

(b) Man berechnet ein Array B der Größe $\mathcal{O}(n^2)$, welches für jedes Paar (i, j) mit $0 \leq j < i < n$ einen Eintrag mit $|A[i]-A[j]|$ enthält. Dies kostet $\mathcal{O}(n^2)$. Jetzt alloziert man eine Hashtabelle der Größe $\mathcal{O}(n^2)$ (womit man sicherstellt dass der Load $\alpha = \mathcal{O}(1)$ ist), wählt eine geeignete Hashfunktion h und hasht die Werte aus B in die Tabelle H (Kosten $\mathcal{O}(n^2)$ unter der Annahme, dass eine insert Operation Kosten $\mathcal{O}(1)$ hat). Anschließend testet man für jeden Wert aus A, ob dieser Wert in H ist, was n mal Kosten $\mathcal{O}(1)$ verursacht. Die Gesamtkosten betragen also $\mathcal{O}(n^2)$.

(c) Man sortiert A (Kosten $\mathcal{O}(n \log n)$). Anschließend berechnet man das Array B wie in Teil (b) (Kosten $\mathcal{O}(n^2)$). Nun testet man für jeden Eintrag in B, ob dieser in A vorkommt, unter Verwendung von binary search. Dies verursacht n^2 mal Kosten $\mathcal{O}(\log n)$. Die Gesamtlaufzeit wird also durch den letzten Schritt dominiert und beträgt $\mathcal{O}(n^2 \log n)$.