Albert-Ludwigs-Universität
Institut für Informatik
Prof. Dr. F. Kuhn
P. Bamberger
P. Schneider

Algorithmen und Datenstrukturen
Sommersemester 2020
Musterlösung Übungsblatt 4

Abgabe: Mittwoch, 10.06.2020, 16:00 Uhr.

Aufgabe 1: Hashing mit offener Adressierung *(10 Punkte)*

Wir betrachten Hashtabellen mit offener Adressierung und zwei Methoden zur Auflösung von Kollisionen: *Lineares Sondieren* und *Doppel-Hashing*. Sei m die Größe der Hashtabelle wobei m Primzahl ist. Sei $h_1(x) := 53 \cdot x$ und $h_2(x) := 1 + (x \bmod (m-1))$. Wir definieren die folgenden beiden Hashfunktionen zur Kollisionsauflösung gemäß der Vorlesung:

- Lineares Sondieren: $h_\ell(x,i) := (h_1(x) + i) \mod m$.
- Doppel-Hashing: $h_d(x,i) := (h_1(x) + i \cdot h_2(x)) \mod m$.

(a) Implementieren Sie eine Hashtabelle mit den Operationen `insert` und `find` gemäß der Vorlesung und den genannten Strategien zur Kollisionsauflösung. Sie *können* dazu die Vorlage `HashTable.py` benutzen. *(5 Punkte)*

(b) Erstellen Sie eine Hashtabelle der Größe $m > 1000$ (m prim) und messen Sie die *durchschnittliche* Laufzeit für das Einfügen von k Schlüsseln für $k \in \{\lfloor \frac{m \cdot i}{50} \rfloor \mid i = 1, \ldots, 49\}$. Wiederholen Sie das Experiment in vier Variationen: Mit linearem Sondieren bzw. Doppel-Hashing; mit k zufälligen Schlüsseln[1] bzw. der vorgegebenen Schlüsselmenge $\{m \cdot i \mid i = 1, \ldots, k\}$. Stellen Sie die durchschnittlichen Laufzeiten für das Einfügen grafisch dar. Diskutieren Sie die Schaubilder in Ihren `erfahrungen.txt`[2]. *(5 Punkte)*

Musterlösung

(a) Siehe Lösungsdatei `HashTable.py`.

(b) Siehe Abbildung 1. Möchte man alle Punktwolken in einem Graphen anzeigen war hier eine logarithmische Skala sehr hilfreich (Abbildung 1, oben), da sich die durchschnittliche Einfügezeiten sehr stark unterscheiden. Wir haben die Variante mit linearem Sondieren und kollidierenden Schlüsseln noch einmal seperat mit linearer Skala dargestellt dargestellt (Abbildung 1, unten).

Am oberen Graph kann man gut erkennen, dass lineares Sondieren zusammen mit einem Input der durch h_1 immer auf den gleichen (0-ten) Tabelleneintrag abgebildet wird, einen worst case für diese Art der Kollissionsbehandlung darstellt. Die durchschnittliche Laufzeit fürs Einfügen nimmt linear zu, was daran liegt dass für das Einfügen des i-ten Schlüssels, i Listenplätze durchlaufen werden müssen, das erkennt man gut am unteren Graphen.

[1] Zufallswerte aus $\{0, \ldots, z\}$ ohne Duplikate mit $z \gg m$. Bspw mit `random.sample(range(z+1), k)`.

[2] Nutzen Sie die `erfahrungen.txt` gerne weiterhin um uns Feedback über Dauer und Schwierigkeiten der Bearbeitung des Übungsblattes mitzuteilen.

Mit randomisierter Eingabe funktioniert lineares Sondieren hingegen gut, zumindest so lange die Tabelle nicht zu voll wird. Wird der Füllstand sehr hoch (wir gehen ja bis c.a. 98%) steigt die durchschnittliche Einfügezeit auch in diesem Fall rapide an (man beachte die logarithmische Skala), da sich mit hoher Wahrscheinlichkeit "Cluster" von besetzten Tabelleneinträgen bilden die eine große Einfügezeit verursachen.

Beim Doppelhashing ist zunächst zu beobachten dass der deterministische Input der beim linearem Sondieren schlecht war, keine Probleme verursacht. Die Laufzeit bleibt für beide Eingabevarianten in etwa gleich, wobei beim randomisierten Input für hohen Füllstand dennoch verstärktes Clustering aufzutreten scheint.

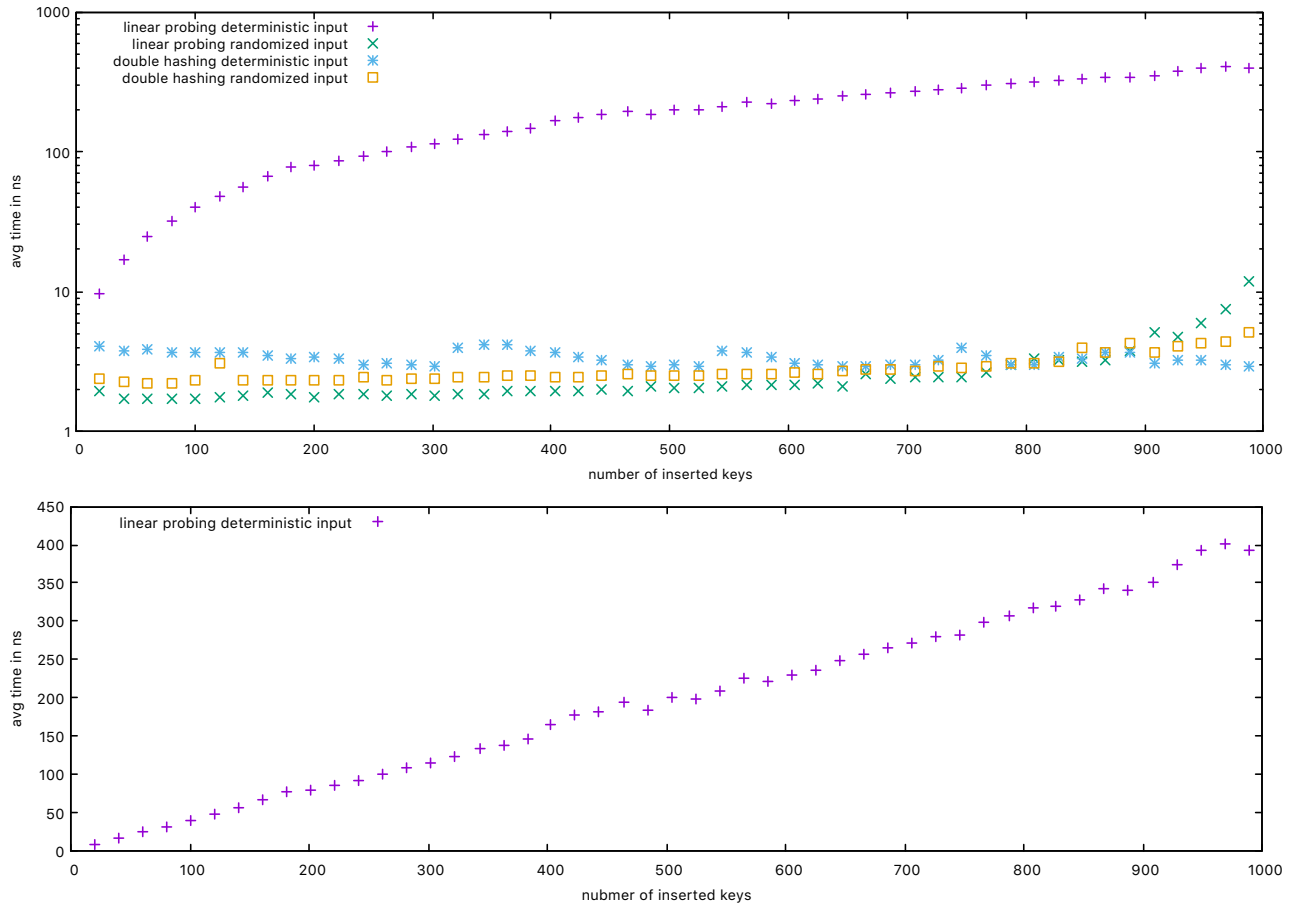

Abb. 1: Plots zu Aufgabe 1 b). Oben: Graph mit allen Varianten und logarithmischer Skala. Unten: Graph mit linearem Sondieren und deterministischem Input.

Aufgabe 2: 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)*

(b) Beschreiben Sie einen alternativen Algorithmus $\mathcal{B}$ für dieses Problem (d.h. $\mathcal{B}(A) = \texttt{algorithm}(A)$ für jede Eingabe A) mit einer Laufzeit von $\mathcal{O}(n^2)$, welcher Hashing verwendet. *(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 $\mathcal{O}(n^2 \log n)$. *(4 Punkte)*

Hinweis: Nutzen Sie Sortierung.

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 $\mathcal{O}(n^2)$. Anschließend alloziiert man eine Hashtabelle der Größe $\mathcal{O}(n^2)$, 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 $\mathcal{O}(n^2 \log n)$.