Algorithmen und Datenstrukturen
WS 2025
von Luxburg/Bordt

Übungsblatt 4

Abgabe bis 10.11.2025 auf Ilias.
Bitte nennen Sie die Abgabe "Blatt4-[Ihr Name].pdf"

Aufgabe 1: Hashing (1 + 1 + 1 + 1 Punkte)

Sei $\mathbb{N}_0 := \{0, 1, 2, \ldots\}$. Wir schreiben $a\mathbb{N}_0 + b := \{a \cdot t + b \mid t \in \mathbb{N}_0\}$ für $a, b \in \mathbb{N}_0$, beispielsweise $5\mathbb{N}_0 + 2 = \{2, 7, 12, 17, \ldots\}$. Geben Sie in dieser Schreibweise jeweils die Menge aller Keys $k \in \mathbb{N}_0$ an (inkl. Begründung), die auf der letzten Position (Index 10) in einer Hashtabelle der Größe 11 kollidieren, wenn folgende Hashfunktionen verwendet werden:

(a) $h(k) = k \mod 11$

(b) $h(k) = 2k \mod 11$

(c) $h(k) = (k^2 + 10) \mod 11$

(d) $h(k) = (3^k - 1) \mod 11$

Tipp: Für (c) und (d) rufen Sie sich die Bedeutung der Primfaktorzerlegung einer natürlichen Zahl in Erinnerung, insbesondere wenn sie quadriert wird.

Aufgabe 2: Hashing mit Verkettung (1+1 Punkte)

Beim Hashing mit Verkettung werden Konflikte dadurch aufgelöst, dass jedes Element i der Hashtafel eine Liste von Werten S_i ist, die alle denselben Hashfunktionswert haben, d.h. $\forall s \in S_i : h(s) = i$.

a) Benutzen Sie die Hashfunktion $h(x) = x \mod 17$ und bilden Sie

$$S = \{19, 17, 38, 51, 25, 26, 77, 34, 99, 1, 7, 85, 4, 31, 91, 68\}$$

auf eine Hashtafel der Größe $m = 17$ mit Verkettung ab.

b) Löschen Sie die Werte 85, 77, 26 und 68 in dieser Reihenfolge aus der Hashtafel. Geben Sie dabei an, wie viele Elemente der entsprechenden Liste jeweils betrachtet werden, bevor der richtige Wert gefunden wird.

Aufgabe 3: Rehashing (0+2+1+1+1 Punkte)

Beim Hashing mit offener Adressierung werden die Elemente in ein Array eingefügt. Auf diese Weise funktionieren z.B. Datenstrukturen wie *std::unordered_map* in C++ oder *dict* in Pyhton. Hier stellt sich das Problem, dass man die Größe des Arrays am Anfang festlegen muss. Der Benutzer der Datenstruktur soll dann aber in der Regel beliebig viele Elemente in die Hashtabelle einfügen können. Die Lösung für dieses Problem ist das sogenannte Rehashing. Beim Rehashing gibt es eine Folge von Hashtafeln $T_0, T_1, \ldots$ so dass die Hashtafel T_i $2^i m$ verschiedene Hashwerte enthält (die erste Hashtafel entspricht dem anfänglichen Array, enthält also m verschiedene Hashwerte). Beim Einfügen und Löschen werden folgende Operationen ausgeführt:

- Wird ein Element in die Tafel T_i eingefügt, wird geprüft, ob nun genau $2^i m$ Elemente in T_i sind, das heißt ob $\alpha = 1$. Ist dies der Fall wird eine Tafel T_{i+1} der doppelten Größe $2^{i+1}m$ erstellt. Diese Operation kostet $\mathcal{O}(2^i m)$ Zeit, weil alle Elemente kopiert werden müssen. Der neue Belegungsgrad ist dann $\alpha = \frac{1}{2}$.
- Wird ein Element aus der Tafel T_i mit $i > 0$ gelöscht, wird geprüft, ob die Tafel T_i genau $2^{i-2}m$ Elemente enthält, das heißt ob $\alpha = 1/4$. Ist dies der Fall wird eine Tafel T_{i-1} der halben Größe $2^{i-1}m$ erstellt. Diese Operation kostet $\mathcal{O}(2^i m)$ Zeit, weil alle Elemente kopiert werden müssen. Der neue Belegungsgrad ist dann $\alpha = \frac{1}{2}$.

Vereinfacht können Sie in dieser Übung annehmen, dass das Einfügen und Löschen in Zeit $\mathcal{O}(1)$ funktioniert (so als würde es zu keinen Konflikten beim Hashing kommen).

a) Bestimmen Sie die worst-case Laufzeit der Einfüge- und Löschoperation.

b) Zeigen Sie, dass eine Folge von n Einfügungen in Zeit $\mathcal{O}(n)$ erfolgen kann.
Sie können davon ausgehen, dass mit einer Tabelle mit Belegungsgrad $\alpha = 1/2$ begonnen wird.

Tipp: Zerlegen Sie die Kosten in Kosten für günstige Einfügungen und Kosten für teure Einfügungen (Tafel wird vergrößert). Beachten Sie dabei, wie viele günstige Einfügungen zwischen der Erstellung von T_{i-1} und der Erstellung von T_i stattfinden.

c) Zeigen Sie, dass eine Folge von n Löschungen in Zeit $\mathcal{O}(n)$ erfolgen kann.
Sie können davon ausgehen, dass mit einer Tabelle mit Belegungsgrad $\alpha = 1/2$ begonnen wird.

d) Zeigen Sie, dass eine Folge von n Operationen (Mischung aus Einfügungen und Löschungen) in Zeit $\mathcal{O}(n)$ erfolgen kann.
Sie können davon ausgehen, dass mit einer Tabelle mit Belegungsgrad $\alpha = 1/2$ begonnen wird.

Tipp: Was passiert, wenn zwischen der Vergrößerung von T_i auf T_{i+1} nicht nur Einfüge- sondern auch Löschoperationen stattfinden ohne dass zwischendrin die Tafel verkleinert wird?

e) Bestimmen Sie die amortisierte Laufzeit der Einfüge- und Löschoperation.

Aufgabe 4: Two Sum (Knobelaufgabe, wird nicht korrigiert, 0 Punkte)

Gegeben sei ein Array von ganzen Zahlen `nums` und eine Zielzahl `target`. Finden Sie zwei verschiedene Indizes i und j im Array, sodass `nums[i] + nums[j] = target`.
Sie können davon ausgehen, dass jede Eingabe genau eine Lösung hat und dass Sie dasselbe Element nicht zweimal verwenden dürfen. Die Rückgabe der Indizes kann in beliebiger Reihenfolge erfolgen.
Beispiel:

```
Eingabe: nums = [2,7,11,15], target = 9
Ausgabe: [0,1]
Erklärung: nums[0] + nums[1] = 2 + 7 = 9
```

a) **Algorithmus:** Beschreiben Sie einen Algorithmus in Pseudocode, der das Problem in Laufzeit $O(n)$ löst, wobei n die Länge des Arrays ist. (Das ist überraschend! Was ist die Laufzeit der brute force Methode? Was wäre die Laufzeit, wenn man Heaps verwendeet? Nun versuchen Sie es mit Hashing ...)

b) **Korrektheit:** Beweisen Sie, dass Ihr Algorithmus korrekt ist. Gehen Sie dabei insbesondere darauf ein, warum Ihr Algorithmus garantiert eine Lösung findet, wenn eine existiert.

c) **Laufzeitanalyse:** Analysieren Sie die Laufzeit Ihres Algorithmus. Begründen Sie, warum die Zeitkomplexität $O(n)$ beträgt.

d) **Implementierung:** Implementieren Sie Ihre Lösung auf LeetCode und reichen Sie sie erfolgreich ein.
`https://leetcode.com/problems/two-sum/`

Aufgabe 5: `heap` and `dict` in Python (Knobelaufgabe, wird nicht korrigiert, 0 Punkte)

`heapq` ist die Standard-Implementierung von min-Heaps in Python. Diese Implementierung unterstützt die drei wichtigsten Operationen, mit denen man eine Priority Queue implementiert: (1) Ein gegebenes Array in einen Heap verwandeln in $O(n)$; (2) Das kleinste Element herausholen in $O(\log n)$; Ein neues Element in den Heap einfügen in $O(\log n)$. Weiterhin gibt es die internen Funktionen `siftup` und `siftdown`, mit denen man die increase/decrease-key Operationen durchführen kann. Aber es gibt eine wichtige Heap-Operationen, die NICHT unterstützt sind: das Löschen eines beliebigen Elementes. Überlegen Sie sich, was eine naive Art ist, das zu implementieren (Sie haben Zugriff auf alle Listenelement, in denen der Heap gespeichert ist). Wie können Sie diese Operation durch Kombination von `heapq` mit einer zusätzlichen Python-Hashtable `dict` mit effizienterer Laufzeit implementieren?