TU Wien
Institut für Logic and Computation
Forschungsbereich Algorithms and Complexity

186.866 Algorithmen und Datenstrukturen VU

Übungsblatt 4

PDF erstellt am: 6. April 2022

Deadline für dieses Übungsblatt ist **Montag, 09.05.2022, 20:00 Uhr**. Um Aufgaben für diese Übung anerkannt zu bekommen, gehen Sie folgendermaßen vor:

1. Öffnen Sie den TUWEL-Kurs der Lehrveranstaltung *186.866 Algorithmen und Datenstrukturen (VU 5.5)* und navigieren Sie zum Abschnitt *Übungsblätter*.
2. Teilen Sie uns mit, welche Aufgaben Sie gelöst haben **und** welche gelösten Aufgaben Sie gegebenenfalls in der Übungseinheit präsentieren können. Gehen Sie dabei folgendermaßen vor:
   - Laden Sie Ihre Lösungen in einem einzigen PDF-Dokument in TUWEL hoch.
     Link *Hochladen Lösungen Übungsblatt 4*
     Button *Abgabe hinzufügen*
     PDF-Datei mit Lösungen hochladen und *Änderungen sichern*.
   - Kreuzen Sie an, welche Aufgaben Sie gegebenenfalls in der Übung präsentieren können. Die Lösungen der angekreuzten Aufgaben müssen im hochgeladenen PDF enthalten sein.
     Link *Ankreuzen Übungsblatt 4*
     Button *Abgabe bearbeiten*
     Bearbeitete Aufgaben anhaken und *Änderungen speichern*.

Bitte beachten Sie:

- Bis zur Deadline können Sie sowohl Ihr hochgeladenes PDF, als auch Ihre angekreuzten Aufgaben beliebig oft verändern. Nach der Deadline ist keine Veränderung mehr möglich. Es werden ausnahmslos keine Nachabgabeversuche (z.B. per E-Mail) akzeptiert.
- Sie können Ihre Lösungen entweder direkt in einem Textverarbeitungsprogramm erstellen, oder aber auch gut leserliche Scans bzw. Fotos von handschriftlichen Ausarbeitungen hochladen (beachten Sie die maximale Dateigröße).
- Beachten Sie die Richtlinien für das An- und Aberkennen von Aufgaben (Details finden Sie in den Folien der Vorbesprechung).

Aufgabe 1. Lösen Sie die folgenden Unteraufgaben zu **B-Bäumen**.

(a) Fügen Sie die Elemente der Folge

$$[2, 5, 7, 11, 16, 22, 24, 29]$$

in dieser Reihenfolge in einen anfangs leeren B-Baum der Ordnung 3 ein. Zeichnen Sie den B-Baum jeweils vor und nach jeder Reorganisationsmaßnahme und geben Sie den endgültigen B-Baum an.

(b) Geben Sie den B-Baum an, der durch Löschen der Schlüssel 16 und 11 (in dieser Reihenfolge) aus dem Baum von Unteraufgabe (a) entsteht. Stellen Sie den B-Baum nach jedem Zwischenschritt dar und führen Sie gegebenenfalls geeignete Reorganisationsmaßnahmen durch.

Anmerkung: Die Angabe von leeren Blättern ist nicht notwendig.

Aufgabe 2. Gegeben seien folgende natürliche Zahlen in der angegebenen Reihenfolge:

$$[5, 16, 9, 4, 20, 0, 3]$$

Fügen Sie diese in der vorgegebenen Ordnung jeweils in folgende Varianten von anfangs leeren Hashtabellen der Größe $m = 11$ ein. Stellen Sie dabei die einzelnen Schritte und die finale Belegung dar. Geben Sie weiters jeweils die durchschnittliche Anzahl der Schritte einer erfolgreichen Suche nach einem Element aus den oben gegebenen Zahlen, sowie die Suchschritte und die Tabelle beim bzw. nach dem Löschen des Schlüssels 16 aus der entstandenen Hashtabelle an.

(a) Verkettung der Überläufer mit $h(k) = k \bmod m$.

(b) Quadratisches Sondieren mit $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$,
$h'(k) = \lceil m(kA - \lfloor kA \rfloor) \rceil$, $c_1 = 3$, $c_2 = 2$ und $A = \sqrt{3}$.

(c) Double-Hashing mit $h(k, i) = (h_1(k) + i h_2(k)) \bmod m$, $h_1(k) = k \bmod m$ und $h_2(k) = (k \bmod 5) + 1$.

Hinweis: Für das Sondieren (quadratisch und Double-Hashing) gilt $i = 0, 1, \ldots, m - 1$.

Hinweis: Bei allen Unteraufgaben können alle Werte eingefügt werden.

Aufgabe 3. Beim sogenannten *Kuckucks-Hashing* benutzt man zwei einfache Hashfunktionen h_1 und h_2 auf zwei Hashtabellen T_1 und T_2. Wollen wir einen Schlüssel k einfügen, so probieren wir dies zunächst beim Index $h_1(k)$ in T_1. Führt dies zu einer Kollision mit einem Schlüssel ℓ, so verschieben wir den Schlüssel ℓ von $h_1(k)$ in T_1 nach $h_2(\ell)$ in T_2 und speichern den Schlüssel k an der Stelle $h_1(k)$ in T_1. Sollte das Verschieben von ℓ wiederum zu einer Kollision mit einem Schlüssel m führen, verschieben wir m von $h_2(\ell)$ in T_2 nach $h_1(m)$ in T_1 und speichern ℓ an der Stelle $h_2(\ell)$ in T_2 ab, und so weiter.

(a) Können nach dem Einfügen mehrerer Elemente in leere T_1 und T_2 beim Kuckucks-Hashing mehr Schlüssel in T_2 gespeichert sein, als in T_1? Begründen Sie Ihre Antwort.

(b) Betrachten Sie Kuckucks-Hashing auf den initial leeren Hashtabellen T_1 und T_2 der Größe 5 mit $h_1(x) = x \bmod 5$ und $h_2(x) = \left\lfloor \frac{x}{5} \right\rfloor \bmod 5$, und fügen Sie in dieser Reihenfolge die Zahlen $[26, 51, 41]$ ein. Geben Sie dabei die Tabellen T_1 und T_2 nach jedem Einfügen an.

(c) Schreiben Sie als Pseudocode eine Methode zur Suche eines Schlüssels beim Kuckucks-Hashing und geben Sie ihre Komplexität in $\mathcal{O}$-Notation an.

(d) Geben Sie *alle* natürlichen Zahlen mit der Eigenschaft an, für die gilt: wenn ein Schlüssel mit dem jeweiligen Wert beim Kuckucks-Hashing Zustand aus Unteraufgabe (b) eingefügt wird, gerät man in eine Endlosschleife.

(e) Überlegen Sie, wie man Endlosschleifen beim Einfügen von Schlüsseln vermeiden kann und welche Auswirkungen Ihre Anpassungen auf die Komplexität der Suche haben.

Aufgabe 4. Geben Sie für die folgenden Fallbeispiele an, welche bisher in der Vorlesung behandelten Datenstrukturen jeweils zum Speichern der fettgedruckten Information am geeignetsten ist. Begründen Sie Ihre Antwort kurz. Beschreiben Sie dabei auch jeweils eine in den Beispielen potentiell wichtige Operation auf den Informationen und deren Komplexität unter Verwendung der von Ihnen angegebenen Datenstruktur.

(a) Der **Zustand eines Schachbretts**, also eines 8×8-Bretts, auf dessen Feldern je bis zu eine Figur stehen kann.

(b) Alle **zulässigen Wörter** für das Spiel Scrabble.

(c) Die *eindeutigen* **Benutzernamen** eines Online-Forums.

(d) Zwei vorgegebene **Mengen von Zahlen**, die man auf Disjunktheit überprüfen möchte.

(e) Zwei vorgegebene **Mengen von Zahlen**, die man auf Mengengleichheit überprüfen möchte.

(f) Globale **Abhängigkeiten** zwischen LaTeX-Paketen.

(g) Eine umfangreiche und dynamische **ToDo-Liste**, die man benutzen möchte, um Aufgaben nach Dringlichkeit zu bearbeiten.

Aufgabe 5. Ein Vertreter der linearen Sortieralgorithmen ist **Radixsort**. Die Grundidee basiert darauf, dass die zu sortierenden Wörter oder Zahlen mehrmals in Fächer (Buckets) verteilt werden, sodass nach der letzten Verteilung die Elemente in sortierter Reihenfolge entnommen werden können.

Finden Sie zunächst mithilfe einer Internet- und/oder Literaturrecherche heraus, wie Radixsort genau funktioniert.

(a) Sortieren Sie die folgenden Zahlen aufsteigend mithilfe von Radixsort:

447, 415, 683, 437, 613, 645, 435

(b) Welches Laufzeitverhalten hat Radixsort?

(c) Handelt es sich bei Radixsort um einen Divide-and-Conquer-Algorithmus? Begründen Sie Ihre Antwort.

(d) Ist Radixsort, so wie Sie es in Unteraufgabe (a) angewendet haben, stabil? Begründen Sie Ihre Antwort.

Aufgabe 6. Gegeben ist ein Array der Größe 20 mit folgenden Zahlen:
4, 8, 9, 13, 16, 12, 2, 19, 14, 1, 18, 15, 11, 7, 3, 20, 5, 10, 6, 17

Sortieren Sie das Array mit **TimSort**. Verwenden Sie dabei die Einstellung Min_Run = 3, sowie lineares Merging ohne Galloping. Geben Sie an, wie Sie die Runs bilden und in welcher Reihenfolge Sie diese mergen. Geben Sie außerdem den Zustand des Run-Stapels nach jeder Änderung an.

Die Zwischenschritte für Insertionsort und das Mergen müssen Sie nicht explizit angeben, sollen diese aber bei Bedarf erklären können.
