Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
Arbeitsgruppe Datenbanksysteme

Sommersemester 2021
16. 05. 2021

Prof. Dr. Bernhard Seeger
Nikolaus Glombiewski M.Sc.
Amir El-Shaikh

Übungen zur Vorlesung
Algorithmen und Datenstrukturen

Abgabe: 23. 05. 2021,
bis **spätestens** 23:59 Uhr
über die ILIAS Plattform

Übung 5

Aufgabe 5.1: Hashtabellen - Verständnis und Algorithmen (2+3) (5 Punkte)

a) Die folgenden Schlüssel wurden in eine anfangs leere Hash-Tabelle der Größe 6 mit Hilfe von linearem Sondieren eingefügt. Die Reihenfolge, mit welcher Schlüssel eingefügt wurden, ist **nicht** bekannt. Das Resultat des Einfügens ist folgende Hash-Tabelle:

Schlüssel	Hash-Wert
E	1
L	5
M	0
N	1
R	5
Z	4

Adresse	0	1	2	3	4	5
Schlüssel	R	M	N	E	Z	L

Geben Sie **alle** möglichen Schlüssel an, welche in der Einfüge-Reihenfolge die letzte Position haben können. Begründen Sie Ihre Antwort für gewählte Schlüssel kurz.

b) Gegeben sei ein Array A der Größe n, welches n Elemente enthält. Beschreiben Sie ein Verfahren, welches für eine Eingabezahl k herausfindet, ob k aus der Summe zweier Elemente in A entstehen kann. Das Verfahren soll eine *erwartete* Laufzeit von $\mathcal{O}(n)$ besitzen. Geben Sie den Algorithmus in Pseudocode an und begründen Sie die erwartete Laufzeit.

Aufgabe 5.2: Implementierung von Hashtabellen (2+2+4+2) (10 Punkte)

In dieser Aufgabe wird ein bereits bekanntes Verfahren für Hashtabellen aus der Vorlesung implementiert und mit einem weiteren Verfahren verglichen werden.

a) Erstellen Sie eine Klasse `LinearProbing`, welche das gegebene Interface `Hashing` gemäß der Strategie *Lineares Sondieren* aus der Vorlesung umsetzt. Übergeben Sie im Konstruktur eine maximale Größe der Hashtabelle und eine `FixedRangeHashFunction`, mit welcher auf diese Größe abgebildet wird. Achten Sie im Konstruktur darauf, dass diese beiden Parameter zueinander passen.

b) Erstellen Sie eine Klasse `RobinHood`, welche das gegebene Interface `Hashing` mit der *Robin Hood Hashing* Strategie umsetzt - ebenfalls mit Hilfe einer maximalen Größe und einer

`FixedRangeHashFunction`. Die Strategie ist analog zum *Linearen Sondieren* ein *geschlossenes Hashverfahren*, bei welchem in Fall von einer Kollision *linear* nach einem Platz in der Hashtabelle gesucht wird. Dabei wird sich zu jedem Element der Hashtabelle die Länge der Sondierungsfolge gemerkt. Im Gegensatz zum *Linearen Sondieren* wird nicht der nächste freie Platz in der Hashtabelle genutzt, sondern das nächste Element mit geringerer Sondierungslänge aus seinem Platz verdrängt. Anschließend wird potentiell rekursiv mit dem verdrängten Element fortgefahren. Dieser Prozess wird fortgesetzt, bis ein Element auf einem freien Platz landet.

c) Erweitern Sie Ihre Klassen `LinearProbing` und `RobinHood` indem Sie das gegebene Interface `Remove` implementieren, welches eine Löschung aus der Hashtabelle erlaubt. Passen Sie dafür ggf. auch Einfüge- und Suchmethoden aus der vorangegangen Aufgabenstellung an. Verwenden Sie in der Klasse `LinearProbing` analog zu in der Vorlesung besprochenen Strategie Markierungen für gelöschte Elemente. Für die Klasse `RobinHood` soll die Hashtabelle während der Löschung direkt aufgeräumt werden. Verschieben Sie dazu nachfolgende Elemente solange in der Sondierungsfolge nach vorne, bis Sie auf ein Elemenet stoßen, was bereits auf seiner Heimatadresse liegt.

d) Stellen Sie für Ihre Implementierung Hypothesen bzgl. Performance-*Unterschieden* der beiden Verfahren auf und halten Sie diese schriftlich fest. Erstellen Sie anschließend eine Klasse *ProbingExperiments* und führen Sie Experimente durch, die Ihre Hypothesen überprüfen. Halten Sie Ihre Ergebnisse schriftlich fest und begründen Sie, warum Ihre Hypothesen in den Experimenten bestätigt oder widerlegt wurden.

Aufgabe 5.3: Verwendung von Hashing* (1+4) **(5 Punkte)**

Ein Puffer kann für eine Datenstruktur verwendet werden, um den Zugriff auf bestimmte Elemente zu beschleunigen. Dabei gibt es verschiedene Strategien. Bei einem Puffer mit der *LRU*-Strategie (Least Recently Used) werden diejenigen Elemente im Puffer gehalten, auf welche zuletzt zugegriffen wurden.

a) Betrachten Sie im Folgenden einen Puffer mit einer Kapazität von 5 Elementen. Führen Sie händisch die unten stehenden Operationen in der gegebenen Reihenfolge auf dem Puffer aus. Zeichen Sie den Zustand des Puffers nach jedem Zwischenschritt. Markieren Sie dabei das zuletzt verwendete Elemente, und das Element, welches, bei fehlendem Zugriff, als nächstes aus dem Puffer entfernt werden würde.

Operationen: put(A), put(H), put(J), get(A), put(E), put(W), put(P), put(A)

b) Vervollständigen Sie die gegebene Klasse `LRUBuffer`, indem Sie die Funktionalität eines Puffers mit LRU-Strategie umsetzen. Die `put` und `get` Methoden sollen hierbei eine *erwartete* Laufzeit von $\mathcal{O}(1)$ besitzen. Ermöglichen Sie das, indem Sie für Ihre Implementierung genau zwei Datenstruktur verwenden: eine Hashtabelle (die in `java.util` verfügbare `HashMap`) und eine doppelt-verkettete Liste (die zur Verfügung gestellte `SimpleDoublyLinkedList`). Testen Sie Ihre Lösung mit einem geeigneten Beispiel.

*** Aufgabe 5.3 ist für Lehramtsstudierende optional.**