Datenstrukturen (DS)

Sommersemester 2015

Dipl-Inf. Bert Besser
Prof. Dr. Georg Schnitger
Hannes Seiwert, M.Sc.

GOETHE
UNIVERSITÄT
FRANKFURT AM MAIN

Institut für Informatik
AG Theoretische Informatik

Übung 5

Ausgabe: 16.06.2015
Abgabe: 30.06.2015

Aufgabe 5.1. *Suchbäume* (3+5 Punkte)

(Vorlesungen am 16.06.2015 und 23.06.2015, Abschnitte 5.1 und 5.2 im Skript)

Wir wollen die Schlüsselmenge $S = \{1, 2, 3, 4, 5, 6, 7\}$ in einem Wörterbuch verwalten.

a) Führen Sie die Operationenfolge `insert(4)`, `insert(2)`, `insert(7)`, `insert(1)`, `insert(3)`, `insert(5)` und `insert(6)` auf einem anfangs leeren *binären Suchbaum* aus und stellen Sie den resultierenden Baum graphisch dar.
Führen Sie anschließend die Operationen `remove(3)` und `remove(4)` aus und stellen Sie wieder den resultierenden Baum graphisch dar.

b) Fügen Sie nun die Schlüsselfolge $1, 4, 2, 5, 3, 7, 6$ in einen anfangs leeren *AVL-Baum* ein. Stellen Sie den Baum inklusive der Balancegrade aller Knoten nach jeder Einfüge-Operation dar. Falls beim Einfügen eine Rotation nötig ist, stellen Sie den Baum zusätzlich auch vor der Rotation dar. Geben Sie außerdem an, welcher Rotationstyp (ZickZick, ZickZack, ZackZick oder ZackZack) verwendet wird.

Aufgabe 5.2. *Binäre Suchbäume* (3+3 Punkte)

(Vorlesung am 16.06.2015, Abschnitt 5.1 im Skript)

Für binäre Suchbäume betrachten wir neben `lookup(`x`)`, `insert(`x`)` und `remove(`x`)` auch die Operation `select(`k`)`, welche den k-kleinsten im Baum gespeicherten Schlüssel bestimmt. (Wenn $x_1, \ldots, x_n$ die im Baum gespeicherten Schlüssel in aufsteigender Reihenfolge sind, dann ist x_k der k-kleinste Schlüssel.)

Dazu erweitern wir die in den Knoten gespeicherte Information: Jeder Knoten v speichert nun zusätzlich die Anzahl der Knoten seines linken Teilbaums.

a) Modifizieren Sie die Funktionen `insert(`x`)` und `remove(`x`)` geeignet, sodass die neuen Informationen stets aktuell bleiben, ohne dass sich deren asymptotische Laufzeit ändert.

b) Implementieren Sie in Pseudocode die Funktion `select(`k`)` in Laufzeit $\mathcal{O}(\text{Tiefe}(T))$, wobei T der Baum ist, auf dem die Operationen ausgeführt werden.

Bitte wenden!

Aufgabe 5.3. *Splitting* (3+5 Punkte)

(Vorlesung am 16.06.2015, Abschnitt 5.1 im Skript)

Wir betrachten binäre Suchbäume, die Mengen von ganzen Zahlen verwalten: $M(T) \subseteq \mathbb{Z}$ bezeichnet die Menge aller im binären Suchbaum T gespeicherten Schlüssel.

Implementieren Sie in Pseudocode die folgenden Funktionen:

a) `join`(T_1, T_2) erhält als Eingabe zwei binäre Suchbäume T_1 und T_2, für die gilt:

$$x < y \text{ für alle } x \in M(T_1) \text{ und für alle } y \in M(T_2)$$

Die Ausgabe von `join` ist ein binärer Suchbaum T, der T_1 und T_2 "vereinigt", für den also $M(T)$=$M(T_1) \cup M(T_2)$ gilt. Außerdem soll $\text{Tiefe}(T) \leq 1 + \max\{\text{Tiefe}(T_1), \text{Tiefe}(T_2)\}$ gelten.

Die Laufzeit soll $\mathcal{O}(\max\{\text{Tiefe}(T_1), \text{Tiefe}(T_2)\})$ nicht überschreiten.

b) `split`(T, x) erhält als Eingabe einen binären Suchbaum T sowie eine Zahl $x \in \mathbb{Z}$.

Die Ausgabe von `split` ist ein binärer Suchbaum T' mit $M(T') = \{y \in M(T) : y \geq x\}$.

Analysieren Sie die Laufzeit Ihres Algorithmus. Die Laufzeit $\mathcal{O}(\text{Tiefe}(T))$ ist möglich.

Hinweis: Bestimmen Sie einen Weg von der Wurzel zu einem Blatt, so dass alle Knoten "links" des Weges nur kleinere Zahlen als x enthalten und alle Knoten "rechts" des Weges nur größere Zahlen als x enthalten. Sie dürfen annehmen, dass $x \notin M(T)$ gilt.

Aufgabe 5.4. *Bin-Packing* (1+1+3+5 Punkte)

(alle bisherigen Vorlesungen bis zum 23.06.2015, Skript bis inkl. Abschnitt 5.2)

Im Bin-Packing-Problem sind n Objekte $1, \ldots, n$ mit den Gewichten $g_1, \ldots, g_n$ für $g_i \in (0, 1]$ gegeben. Die Objekte sind in möglichst wenige Behälter zu verteilen, wobei jeder Behälter Kapazität 1 hat. Jedes Objekt soll also genau einem Behälter zugewiesen werden, wobei der Füllstand eines jeden Behälters die Kapazität 1 nicht übersteigen darf: Wenn in Behälter B die Objekte der Menge $I \subseteq \{1, \ldots, n\}$ enthalten sind, dann ist der *Füllstand* von B gleich $f_B = \sum_{i \in I} g_i$. Die *Restkapazität* von B ist natürlich $r_B = 1 - f_B$.

Vorab ist nicht bekannt wie viele Behälter schließlich benötigt werden. Insbesondere ist die Anzahl der benötigten Behälter mindestens 1 und höchstens n.

Wir betrachten einfache Bin-Packing-Algorithmen, welche die Elemente in der Reihenfolge $1, \ldots, n$ nacheinander den Behältern zuweisen. Dabei darf ein Objekt i nur dann in einen Behälter B eingefügt werden, wenn dieser noch genug Restkapazität hat, d.h. wenn $r_B \geq g_i$ gilt. Nur wenn unter den bereits geöffneten Behältern keiner mit ausreichender Restkapazität vorhanden ist, wird ein neuer Behälter geöffnet. Hier eine schematische Implementierung in Pseudocode:

```
initialisiere die leere Datenstruktur D        // D wird alle geöffneten Behälter verwalten
für i = 1...n
    wenn in D ein "geeigneter" Behälter B mit Restkapazität mindestens g_i existiert,
        dann entferne B aus D
    sonst
        B = neuer Behälter
    füge Objekt i in Behälter B ein        // die Restkapazität von B verringert sich um g_i
    füge B in die Datenstruktur D ein
```

Beachten Sie, dass noch unspezifiziert ist, *welcher* Behälter aus D entfernt wird, falls mehrere geeignete Behälter in D enthalten sind.

Bitte wenden!

Wir betrachten zwei verschiedene Auswahlstrategien:

- Die *Worst-Fit*-Strategie wählt unter allen in D enhaltenen Behältern denjenigen mit **größter** Restkapazität, in den Objekt i noch hineinpasst.
- Die *Best-Fit*-Strategie wählt unter allen in D enhaltenen Behältern denjenigen mit **kleinster** Restkapazität, in den Objekt i noch hineinpasst.

a) Zum warm werden: Verteilen Sie die Objekte $1, \ldots, 6$ mit den Gewichten g_1=0.4, g_2=0.8, g_3=0.7, g_4=0.1, g_5=0.3, g_6=0.5 unter Verwendung von

   i) Worst-Fit und
   ii) Best-Fit.

   Es genügt, das Ergebnis anzugeben.

b) Wir wollen beide Strategien so implementieren, dass alle n Objekte in Zeit $\mathcal{O}(n \log n)$ verteilt werden. Jeweils muss also die Datenstruktur D passend gewählt werden: Wir müssen D schnell nach einem geeigneten Behälter durchsuchen können, ihn schnell entfernen können und einen Behälter schnell einfügen können. (Z. B. ist die Verwendung linearer Listen unpassend, da lineare Suchzeit $\Omega(n)$ nötig ist und quadratische Laufzeit $\Omega(n^2)$ folgt.)

   Beschreiben Sie jeweils eine passende Datenstruktur für

   i) Worst-Fit und
   ii) Best-Fit.

   Erläutern Sie auch, warum die geforderte Laufzeitschranke eingehalten wird.

   Hinweis: Es ist in dieser Aufgabe nicht nötig, die einem Behälter zugewiesenen Objekte zu speichern!