Albert-Ludwigs-Universität, Inst. für Informatik
Prof. Dr. Fabian Kuhn
Y. Maus, O. Saukh

03 Juni, 2016

Algorithmen und Datenstrukturen
Sommersemester 2016
Übungsblatt 6

Abgabe bis 12:00, Freitag, 10 Juni, 2016

Achtung: Bitte fügen Sie Ihrer Lösung eine (kurze) Datei *erfahrungen.txt* mit Ihren Erfahrungen mit dem jeweiligen Aufgabenblatt hinzu. Das Forum können (und sollen) Sie nicht nur für Fragen zur Übung, sondern auch für Fragen zur Vorlesung benutzen.

Aufgabe 1: BSTree, durchschnittliche Tiefe (4 Punkte)

Im *public* SVN finden Sie ein Modul *BSTree.py* welches zwei Klassen enthält - *Node* und *BSTree*. Die erste Klasse representiert Knoten eines binären Suchbaumes und die zweite Klasse ist eine Implementierung eines binären Suchbaumes.
Implementieren Sie eine Funktion *avg_depth()* in der Klasse *BSTree*, welche die **durchschnittliche Tiefe** aller Knoten des Baumes berechnet. Die Tiefe eines Knotens ist als Distanz des Knotens zur Wurzel definiert. Die Wurzel hat Tiefe 0.

Hinweis: Die Programmiervorlagen für die Klassen BSTree und Node stehen auch in C++ und Java zur Verüfung.

Aufgabe 2: Treap (11 Punkte)

Bauen Sie die Klassen *BSTree* und *Node* so um, dass aus dem binären Suchbaum ein **Treap** wird. Dazu fügen Sie in der Klasse **Node** ein neues Feld ein, welches den zweiten immer zufällig gewählten Schlüssel repräsentiert (in der Vorlesung heißt dieser *key2*). Modifizieren Sie alle Operationen in der Klasse **BSTree** so, dass sie für die Klasse *Treap* funktionieren.

Hinweis: Sie können entweder die bestehenden Klassen erweitern (Vererbung) oder den Code umschreiben.

Aufgabe 3: BFS Range (5 Punkte)

Implementieren Sie eine Funktion *bfs_range(a,b)*, die alle Schlüssel, welche im Treap im Bereich zwischen a und b liegen, in der BFS-Traversierungsreihenfolge ausgibt. Die Funktion soll Zeitkomplexität $\mathcal{O}(T+R+1)$ haben, wobei T die Tiefe des Treaps und R die Anzahl der Elemente der Datenstruktur im Bereich ist.
Da die Struktur des Treaps randomisiert erstellt wird und somit die Ausgabe von *bfs_range(a,b)* nicht ausschließlich von den eingefügten Elementen und der Einfügereihefolge abhängt, ist das Testen etwas schwierig. Daher wird eine Funktion *simple_bfs_range(a, b)* zur Verfügung gestellt, welche die Aufgabe naiv in Zeit $\mathcal{O}(n)$ löst. Zum Testen vergleichen Sie Ihre Ausgabe bitte mit dem Ergebnis dieser Funktion.

Bemerkung: Falls Sie die zweite Aufgabe nicht gelöst haben, implementieren Sie bfs_range für den binären Suchbaum.