Lehrstuhl Algorithmen
und Komplexität
Prof. Dr. Fabian Kuhn
Sebastian Daum, Oleksii Saukh

Algorithmen und Datenstrukturen
(Informatik II) SS 2014

http://ac.informatik.uni-freiburg.de/teaching

UNI FREIBURG

Übungsblatt 6

Abgabe: Dienstag, 24. Juni, 16:00 Uhr

Laden Sie Ihre Lösungen für theoretische *Übungen im Format "loesung-xx-y.pdf" hoch, wobei "xx" die Nummer des aktuellen Übungsblattes ist (bei Bedarf mit führender Null) und "y" die Aufgabennummer, d.h., erzeugen Sie* pro (theoretische) Aufgabe *eine* .pdf*-Datei. Scans, Bilder von Scans etc. bitte zuerst ins pdf-Format umwandeln. Bei Programmieraufgaben müssen Sie – wann immer es relevant/durchführbar ist – Unit Tests sowie einen Style-check durchführen. Laden Sie außerdem eine Datei "erfahrungen.txt" hoch, in welcher Sie Ihre Erfahrungen und Meinungen zu dem Übungsblatt teilen.*

Aufgabe 1 (10 Punkte)

In der Vorlesung haben Sie *Treaps* kennengelernt. Kurze Wiederholung: Beim Einfügen eines Elements e mit Schlüssel $e.key_1$ wird ein zweiter Schlüssel $e.key_2$ zufällig erzeugt. Danach wird e in den Treap T eingefügt, so dass die Datenstruktur bezüglich $e.key_1$ ein binärer Suchbaum ist, und anschließend so herumgeschoben und der Treap abgeändert, so dass T bezüglich $e.key_2$ ein *Heap* ist, d.h., die Schlüssel von Vaterelementen sind immer kleiner als die von Kindern.

Vervollständigen Sie für die Operation *delete(x)* den unten gegebenen Pseudo-Code, wobei x der *Schlüssel* $e.key_1$ des zu löschenden Knotens e ist. Als gegeben dürfen Sie die Operationen *leftRotate(p, c_r)*, *rightRotate((p, c_l))*, *predecessor(u)* und *successor(u)* betrachten; siehe Kommentare im Code.

Führen Sie Anschließend den formalen Beweis, warum die beschriebene Struktur tatsächlich funktioniert und die Treap-Datenstruktur erhält. Sie können auf Aussagen, die Sie im Zusammenhang mit binären Suchbäumen kennen, ohne Beweis zurückgreifen.

Hinweis: Die unten stehende Operation *deleteNode* ist nicht zwingend zu verwenden, kann aber als Hilfsmittel dienen. Wenn Sie diese Operation verwenden, müssen Sie allerdings auch den Pseudo-Code dafür angeben.

function DELETE(x)
// Löscht den Knoten e mit $e.key_1 = x$, falls vorhanden; gibt nichts zurück, verändert nur T
auszufüllen

function DELETENODE(u)
// Löscht den Knoten u, wobei u ein Knoten mit maximal einem Kind sein muss; gibt nichts zurück, verändert nur T
optional; müssen Sie ausfüllen, wenn Sie diese Prozedur verwenden wollen

function PREDECESSOR(u)
// Findet den *predecessor* (Vorgänger) v von u bzgl key_1 in u's linkem Teilbaum; u muss ein Knoten mit linkem Kind sein; gibt v zurück
gegeben

function SUCCESSOR(u)
// Findet den *successor* (Nachfolger) v von u bzgl key_1 in u's rechtem Teilbaum; u muss ein Knoten mit rechtem Kind sein; gibt v zurück
gegeben

function LEFTROTATE((p, c_r))
// Macht eine Linksrotation, wobei c_r zu Beginn der Operation das rechte Kind von p ist; gibt nichts zurück,

verändert nur T

gegeben

function RIGHTROTATE((p, c_l))

// Macht eine Rechtsrotation, wobei c_l zu Beginn der Operation das linke Kind von p ist; gibt nichts zurück, verändert nur T

gegeben

Aufgabe 2 (10 Punkte)

In dieser Aufgabe sollen Sie eine Datenstruktur T implementieren, welche die Häufigkeit von Elementen in T speichert und gleichzeitig die Elemente sortiert hält.

Im *public* Verzeichnis gibt es bereits vorgefertigte binäre Suchbäume aus dem letzten Übungsblatt. Passen Sie diese wie folgt an. Die Datenstruktur soll auch ein binärer Baum sein. Des Weiteren soll sie *anstatt* von *insert(e)*, *remove(e)* und *find(e)* die Operationen *increment(e)*, *decrement(e)* und *intervalCount(l,r)* unterstützen, wobei die einzufügenden Werte e gleichzeitig auch die Schlüssel sind. Jeder Wert e wird in T als Tupel $x = (key, c, C)$ abgespeichert, wobei $key = e$, c ist die Anzahl, wie oft e in T enthalten ist und C ist die Anzahl aller Werte im Unterbaum von x, x miteingerechnet.

increment(e) sucht den Wert e in T, und wenn es fündig wird beim Knoten x, dann erhöht es $x.c$ um eins. Wird es nicht fündig, dann wird ein neuer Knoten x eingefügt mit $x.key = e$, $x.c = 1$ und passendem $x.C$. Je nachdem wo notwendig, muss bei allen relevanten Knoten y das Attribut $y.C$ angepasst werden.
decrement(e) ist die passende Gegenoperation. Wird e nicht in T gefunden, so soll nichts passieren, andernfalls wird beim passenden Knoten $x.c$ um eins reduziert. Sinkt bei diesem Update der Wert von $x.c$ auf 0, so soll x komplett aus der Datenstruktur entfernt werden. Auch hier soll wieder bei allen relevanten Knoten y der Wert $y.C$ angepasst werden, falls notwendig.
intervalCount(l,r) bekommt als Eingabe zwei Werte l und r, welche mit den Schlüsseln in T vergleichbar sind, d.h., Tests wie $l < x.key$ sind immer möglich. Bei Aufruf der Funktion soll die Anzahl aller Werte e in T wiedergegeben werden, für die $l \leq e \leq r$ gilt, d.h.,

$$\textit{intervalCount(l,r)} = \sum_{x \in T:\, l \leq x.key \leq r} x.c.$$

Weder l noch r müssen dabei in T als Schlüssel vorkommen.

Stellen Sie sicher, dass Operationen *increment* und *decrement* in Laufzeit $O(d)$ implementiert sind (d ist die Tiefe des Baums).

Im *public* Verzeichnis finden Sie eine Datei *input.txt*, welche Wörter enthält, welche als Schlüssel dienen sollen (die Datei hat die gleiche Strucktur wie für das Übungsblatt 5). Lesen Sie die Wörter Schritt für Schritt ein und fügen Sie diese mit *increment* in Ihre Datenstruktur ein. Führen Sie dann folgende Befehle in der angegebenen Reihenfolge aus und messen Sie die Laufzeiten:

1. *intervalCount("adf","vxz")*,
2. *intervalCount("unb", "une")*,
3. *intervalCount("error","error")*,
4. 20x *delete("underwhelm")*,
5. 20x *delete("uncle")*,
6. *intervalCount("adf","vxz")*,
7. *intervalCount("unb", "une")*,

Erstellen Sie eine Datei *output.txt*. Die Datei muss sieben Zeilen enthalten - einen für jede Testaufgabe. In jeder Zeile muss die benötigte Laufzeit stehen, sowie, falls *intervalCount* ausgeführt wurde, der zurückgegebene Wert.

Hinweis: IntervalCount, wenn geschickt und korrekt implementiert, hat auch nur eine Laufzeit von $O(d)$. I.A. sollten die Laufzeiten Ihrer Testläufe äußerst gering sein.