Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Department Informatik
Markus Püschel
Peter Widmayer
Thomas Tschager
Tobias Pröger
Tomáš Gavenčiak

1. Dezember 2016

Algorithmen & Datenstrukturen — Blatt P11 — HS 16

Abgabe: Bis zum 8. Dezember 2016 um 10 Uhr auf dem Judge (ausschliesslich Quellcode).

Aufgabe P11.1 *Binäre Suchbäume.*

Ein binärer Suchbaum ist ein gewurzelter Baum, bei dem jeder Knoten X einen *Schlüssel* mit Wert $\text{key}(X)$ speichert und ein *linkes Kind* $\text{left}(X)$ und/oder ein *rechtes Kind* $\text{right}(X)$ haben kann. Sei $S(X)$ die Menge aller Schlüssel im Teilbaum mit Wurzel X (inklusive $\text{key}(X)$). Es gilt folgende wichtige Regel: In einem Suchbaum sind alle Werte in $SN(\text{left}(X))$ kleiner als $\text{key}(X)$ und alle Werte in $S(\text{right}(X))$ sind grösser als $\text{key}(X)$. In dieser Aufgabe sind der Einfachheit halber alle Schlüssel verschieden. Ein Elternknoten $\text{parent}(X)$ ist ein Knoten, sodass X entweder das linke oder das rechte Kind dieses Knoten ist.

In der Vorlesung haben Sie AVL Bäume kennengelernt. Diese sind *balancierte* binäre Suchbäume. Es wird nicht nur die Suchbaum-Regel garantiert, sondern zusätzlich, dass der Baum nicht sehr tief ist. In dieser Aufgabe sollen sie verschiedene Operationen für binäre Suchbäume implementieren *ohne* zu rebalancieren. In sehr tiefen, unbalancierten Bäumen können die Operationen sehr aufwendig werden, aber für unsere Eingabedaten ist garantiert, dass höchstens eine Tiefe 50 erreicht wird. Wegen dieser Annahme empfehlen wir, dass Sie rekursive Funktionen nutzen, um die Operationen zu implementieren.

Wir stellen für diese Aufgabe eine Programmvorlage als Eclipse Projektarchiv auf der Vorlesungswebseite zur Verfügung. In der Vorlage sind alle nötigen Funktionen für das Einlesen der Eingabe und für die Ausgabe, sowie eine Klasse `TreeNode` bereits implementiert. Sie müssen lediglich die fehlenden Funktionen implementieren und sie für Teilaufgabe B leicht verändern. Weiter unten werden einige Details zum Design der Klasse `TreeNode` beschrieben.

Teilaufgabe A In der ersten Teilaufgabe sollen Sie die Funktionen `insert`, `contains` und `delete` implementieren. Beachten Sie die Beispiele unten.

`TreeNode.insert(val)` fügt den Schlüssel *val* als ein neuer *Blattknoten*[1] ein ohne den Rest des Baumes zu verändern[2]. Sie dürfen annehmen, dass *val* vor dem Aufruf der Funktion nicht bereits im Baum gespeichert ist.

`TreeNode.contains(val)` überprüft ob *val* im Baum gespeichert ist ohne den Baum zu verändern.

`TreeNode.delete(val)` entfernt den Schlüssel *val* wie folgt aus dem Baum. Sei X der Knoten mit Schlüssel $\text{key}(X) = val$. Wenn X ein Blatt ist, wird er einfach von seinem Elternknoten entfernt. Falls X genau einen Kind Y (linkes oder rechtes) hat, ersetzen wir X durch Y in

[1] Ein *Blatt* ist ein Knoten ohne Kinder.

[2] Beachten Sie, dass es immer eine eindeutige Position gibt, an welcher der Schlüssel eingefügt werden soll.

parent(X) (und entfernen dadurch X). Falls X zwei Kinder hat, sei Y der Knoten mit kleinstem Schlüssel, der *grösser* ist als key(X). Beachten Sie, dass Y immer in S(right(X)) ist. Wir löschen Y aus dem Baum (indem wir rekursiv key(Y) aus dem Teilbaum von X entfernen) und setzen dann key(X) = key(Y) (wir bewegen den Schlüssel von Y zu X, sodass nur der Schlüssel von X gelöscht wird). Sie können annehmen, dass der gelöschte Schlüssel vor dem Aufruf im Baum gespeichert ist.

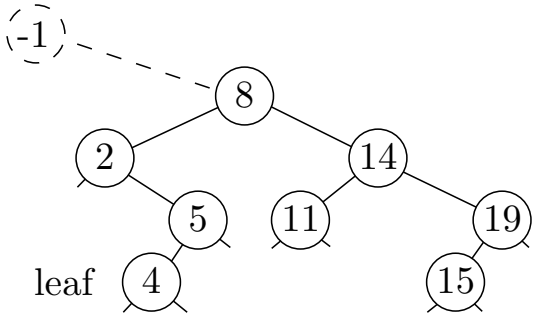
Teilaufgabe B Verändern Sie die Klasse `TreeNode`, damit zusätzlich die Grösse des Teilbaums eines Knoten gespeichert wird, d.h. speichern Sie size(X) = $|S(X)|$. Verändern Sie `insert` und `delete`, sodass die beiden Funktionen die Grösse der Teilbäume richtig aktualisieren. Beachten Sie, dass nur die Grösse derjenigen Teilbäume geändert wird, die sich auf dem Pfad von dem hinzugefügten oder gelöschten Knoten zur Wurzel befinden. Die Grösse ändert sich nur um +1 oder -1.

Implementieren Sie schliesslich die Funktion `TreeNode.findByRank(r)`, die den r-kleinsten Schlüssel im Baum zurückgibt, wobei wir beim Zählen mit 0 beginnen. `findByRank(0)` gibt also das kleinste Element zurück.

Eine Möglichkeit, um den r-kleinsten Schlüssel in einem Teilbaum von X zu finden, ist: Falls $k <$ size(left(X)) dann ist der gesuchte Schlüssel auch der r-kleinste Schlüssel im Teilbaum von left(X) und wir suchen rekursiv dort weiter. Falls $k =$ size(left(X)), ist das gesuchte Element key(X). Falls $k >$ size(left(X)) gilt, ist der gesuchte Schlüssel das $(k -$ size(left(X)) $- 1)$-kleinste Element im Teilbaum von right(X).

Beispiel

Wir fügen die folgenden Schlüssel in der gegebenen Reihenfolge in einen leeren Suchbaum ein: 8, 2, 14, 5, 11, 19, 15, 4. Der Baum, der dadurch entsteht, sieht wie folgt aus und entspricht in Klammer-Notation `(2 ((4) 5)) 8 ((11) 14 ((15) 19))` (wir verwenden den Schlüssel -1 als Wurzel – siehe Details zur Implementierung weiter unten). Beispiele für die Funktionen `delete` und `findByRank` finden Sie weiter unten.



Ein- und Ausgabe

Die Eingabe besteht aus mehreren Befehlen. Jeder Befehl besteht aus einem Buchstaben und optional einer Zahl, durch Leerzeichen oder Zeilenumbrüche getrennt. (Die Eingabedaten am Judge bestehen aus einem Befehl pro Zeile, die Vorlage kann allerdings jede Kombination von Leerzeichen und Zeilenumbrüchen lesen). Einige Befehle geben eine einzige Zeile als Ausgabe aus. Das Programm startet mit einem leeren Baum.

`I val` fügt einen Schlüssel *val* in den Baum ein. `C val` überprüft ob Schlüssel *val* im Baum gespeichert ist und gibt dann entsprechend `YES` oder `NO` aus. `D val` löscht Schlüssel *val* aus dem Baum. `P` gibt den Baum in Klammer-Notation aus (siehe Beispiel weiter unten; die Funktion für

die Ausgabe ist bereits implementiert). `R rank` gibt den *rank*-kleinsten Schlüssel des Baumes aus (beim Zählen mit 0 beginnend). `X` beendet das Programm.

Alle Schlüssel sind Ganzzahlen zwischen 0 und 10 000 000 und es sind höchstens 100 000 Schlüssel im Baum gespeichert. Es werden höchstens 200 000 Operationen durchgeführt. Die Tiefe des Baumes ist kleiner als 50.

Bonus Sie erhalten einen Bonuspunkt pro 100 Punkte auf dem Judge (abgerundet). Insgesamt können Sie bis zu 200 Punkte auf dem Judge erhalten. Damit alle Tests auf dem Judge erfolgreich sind, sollte die Laufzeit jeder Operation (mit Ausnahme von `P`) in $\mathcal{O}(d)$ liegen, wobei d die Tiefe des Baumes ist.

Senden Sie Ihr `Main.java` unter folgendem Link ein: `https://judge.inf.ethz.ch/team/websubmit.php?cid=18985&problem=DA_P11.1`. Das Passwort für die Einschreibung ist "`quicksort`".

Die Testdaten `test1` und `judge1` (30 Bonuspunkte) enthalten lediglich `insert`, `print` und `contains` Befehle, `test2` und `judge2` (70 Bonuspunkte) zusätzlich `delete` Befehle und `test3` und `judge3` (100 Bonuspunkte) enthalten alle möglichen Befehle.

Beispiel

Eingabe (mehrere Befehle pro Zeile) und Ausgabe (des letzten Befehls)

```
I 8 I 2 I 14 I 5 I 11 P        | (2 (5)) 8 ((11) 14)
C 5                            | YES
I 19 I 15 C 4                  | NO
I 4 C 4                        | YES
C 3                            | NO
P                              | (2 ((4) 5)) 8 ((11) 14 ((15) 19))
D 2 D 5 C 4                    | YES
P                              | (4) 8 ((11) 14 ((15) 19))
D 4 D 14 D 11 C 15             | YES
P                              | 8 (15 (19))
I 2 I 5 P                      | (2 (5)) 8 (15 (19))
R 1                            | 5
R 3                            | 15
R 4                            | 19
X
```

Hinweise Der Baum wird durch eine einzige Klasse `TreeNode` implementiert. Ein fehlendes linkes oder rechtes Kind soll `null` sein (Nullzeiger). Dasselbe gilt für den fehlenden Elternknoten der Wurzel. Üblicherweise implementiert man eine Klasse `BinaryTree`, welche den Baum repräsentiert, und die Knoten nicht zugänglich macht. Für diese einfache Aufgabe ist das allerdings nicht nötig.

Unser Baum hat immer eine Wurzel mit einem speziellen Schlüssel -1 (d.h. immer kleiner als alle anderen Schlüsse in der Eingabe) und alle Operationen werden auf die Wurzel angewandt. Das vereinfacht viele der Operationen, da man sonst für einige Operationen den Spezialfall des leeren Baumes beachten müsste oder das Verändern der Wurzel (z.B. hat die Wurzel keinen Elternknoten). Das ist hier nicht nötig, da diese spezielle Wurzel nie verändert wird und nie nach dem Schlüssel -1 gesucht wird.

Reminder: Vergessen Sie nicht, den Elternknoten `parent` jedes Knoten, den Sie verändern, zu aktualisieren.

Das Projektarchiv auf der Vorlesungswebseite enthält weitere Testdaten, damit Sie lokal testen können. Ausserdem stellen wir zusätzlich ein `Judge.java` Programm zur Verfügung, das Ihr Programm `Main.java` mit allen verfügbaren Testdaten testet – öffnen und starten sie dazu einfach `Judge.java` in derselben Weise wie Sie `Main.java` starten würden. Die Art und Weise, wie Ihr Programm `Main.java` arbeitet, wird dadurch nicht beeinflusst. `Judge.java` soll lediglich das lokale Testen erleichtern. Die zur Verfügung gestellten Testdaten sind nicht die Testdaten, welche der Judge verwendet, und im Vergleich nicht so umfangreich.