Universität Konstanz
Fachbereich Informatik & Informationswissenschaft
Prof. Dr. Ulrik Brandes / Dr. Andreas Karrenbauer / Dr. Jürgen Lerner / Martin Mader

Algorithmen und Datenstrukturen
WS 2010/2011

8. Übungsblatt

Ausgabe: 08. Dezember 2010 **Abgabe:** 15. Dezember 2010, 10 Uhr
Die Bearbeitung in Zweiergruppen ist ausdrücklich erwünscht.

Aufgabe 1: Rot-Schwarz-Bäume — 5 Punkte

(a) Fügen Sie in einen anfänglich leeren Rot-Schwarz-Baum nacheinander die Werte $41, 38, 31, 12, 19, 8$ ein und skizzieren Sie seinen Zustand nach jeder Einfügung (natürlich auch die Farben der Knoten).

(b) Löschen Sie aus dem in (a) entstandenen Rot-Schwarz-Baum die Werte $8, 12, 19, 31, 38, 41$ und skizzieren Sie seinen Zustand nach jeder Löschung.

Aufgabe 2: B-Bäume — 5 Punkte

(a) Fügen Sie die Buchstaben $F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E$ in einen anfänglich leeren B-Baum mit $d = 4$ ein. Skizzieren Sie den Baum jedes Mal, wenn ein neuer Knoten entsteht, und am Ende.

(b) Löschen Sie aus dem in (a) entstandenen B-Baum der Reihe nach die Buchstaben C, P und V. Skizzieren Sie nach jedem Löschvorgang den Baum.

Aufgabe 3: AVL-Bäume — 7 Punkte

Realisieren Sie einen AVL-Baum, indem Sie eine Klasse `u08.name.AVLTree` von der Klasse `BinarySearchTree` aus der letzten Übung ableiten (Sie dürfen auch den entsprechenden Lösungsvorschlag verwenden). Überschreiben Sie dabei nur die Methode `update(BinaryTreeNode node)`; es sollen aber weitere private Methoden verwendet werden, um den Code lesbarer zu gestalten.

Zur Vereinfachung des Codes und zur besseren Testbarkeit können Sie folgendes annehmen:

- Die Höhe eines "leeren" Teilbaums ist -1.
- Sie dürfen in `update` auch nach einer Einfügeoperation immer komplett bis zur Wurzel aufsteigen.
- Die in der Rebalancierung involvierten Knoten v und w (Bezeichnung laut Skript) können auch nach einer Einfügeoperation wie nach einer Löschoperation durch

  $$\begin{array}{ll} v & \text{Kind von } u \text{ mit größerer Höhe} \\ w & \text{Kind von } v \text{ mit größerer Höhe} \end{array}$$

  bestimmt werden. (Überlegen Sie sich, wieso dies gilt!)

  Bei gleicher Höhe soll das Kind aber nicht beliebig gewählt werden: Falls etwa die zwei Kinder $v.left$ und $v.right$ eines Knotes v diesselbe Höhe haben, wird das Kind w durch

  $$w = \begin{cases} v.left & \text{falls } v = (v.parent).left \text{ oder } v = root \\ v.right & \text{sonst} \end{cases}$$

  festgelegt.

Aufgabe 3: Erweitertes `find`  3 Punkte

Leiten Sie eine Klasse `u08.name.ExtendedSearchTree` von der Klasse `BinarySearchTree` oder der Klasse `AVLTree` ab, und fügen Sie eine Methode

```
public void findAll(java.util.List<E> list, K min, K max)
```

hinzu. Diese soll in die übergebene Liste alle Elemente schreiben, die zu Knoten v gehören, für die $min \leq v.key \leq max$ gilt. Die Reihenfolge der Elemente in der Liste soll der in-order-Reihenfolge der zugehörigen Knoten entsprechen.

Achten Sie auf eine effiziente Implementation. `findAll` ist in $\Theta(h + s)$, wobei h die Höhe des Suchbaums und s die Anzahl der gefundenen Knoten bezeichnet.