CSci 242: Algorithms and Data Structures **Fall, 2019**

Instructor: Dr. M. E. Kim Date: September 27th, 2019

Due: by the end of day, October 7th (Mon.), 2019.

**Home Assignment 3: 60 points + 140 (implementation) + 10 (Optional)**

Q1. [10] **Balanced Binary Search Tree**

Write an algorithms ***min*(T)** and ***max*(T)**, which return the key-value pair (*k*, *e*) with *smallest* and *largest* key, respectively, in O(log *n*) time in the *balanced binary search tree, T*.

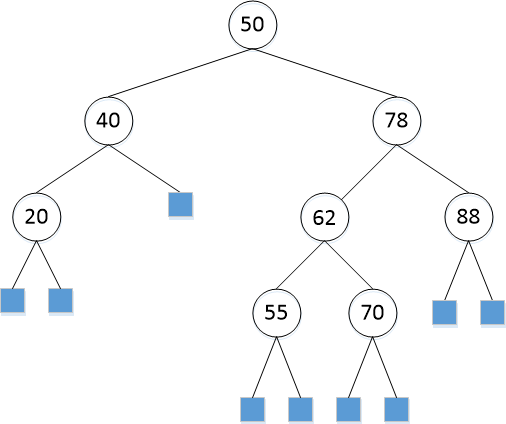
Q2. [10] **AVL Tree**

Consider the insertion of the following keys (in the given order) into an initially empty AVL tree:

15, 45, 55, 35, 25, 40, 10. Draw the final AVL tree.

Q3. [10] **AVL Tree**

Draw the AVL tree resulting from the removal of the key 40 from the AVL tree of Figure below.



Q3B. [70]

For Q2 – Q3, implement the operations (insert, remove, restructure, etc.) of AVL tree in Python or in Java. Print the resulting tree from the root in the following format:

(a key of node: a depth of node, a key of parent, a key of left-child, a key of right-child)

e.g.) (50: 0, null, 40, 78)

Since AVL tree is a balanced Binary Search Tree, you may be able to reuse your Java codes of certain operations from HW 2.

Include the image of your output to the corresponding question.

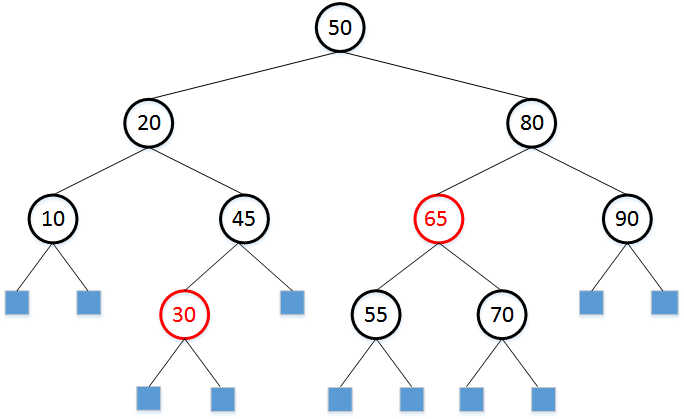
Q4. [10] **AVL Tree**

Professor Amadeus claims that the order of keys to be inserted into an AVL tree does not matter; i.e. the same tree always results every time. Give a counter example that contradicts Prof. Amadeus’s claim.

Q5. [Optional, 10] **AVL Tree**

Draw an example of an AVL tree such that a single remove operation could require Θ(log *n)* trinode restructurings (i.e. rotations) from a leaf to the root in order to restore the height-balance property.

Q6. [20] **Red-Black Tree**



1. [10] Draw the Red-Black tree after each insertion of 35 and 40 into the given RB-tree above.
2. [10] In the given Red-Black Tree above, draw the resulting Red-Black tree after deleting 10..

Q6B. [70] Similarly, implement the operations of Red-Black tree in Q6. Print the resulting tree in the form of:

((a key of node, color), depth, (a key of parent, color), (a key of left-child, color), (a key of right-child, color))

e.g.) ((80, B), 1, (50, B), (65, R), (90, B))

Since RB-tree is a balanced Binary Search Tree, you may be able to reuse your Python/Java codes of certain operations from HW 2.

Include the image of your output to the corresponding question.