CS3610 Project 2

Dr. Jundong Liu

Due: Oct. 2, Friday, 11:59 pm

A binary search tree is a data structure designed for efficient item insertion, deletion, and retrieval. These 3 operations share an average run time complexity of $O(\log(n))$. Such time complexities are guaranteed whenever you are working with a balanced binary search tree. However, if you have a tree that begins leaning heavily to either its left or right side, then you can expect the performances of the insertion, deletion, and retrieval operations to degrade. As an example, consider a scenario in which 5 nodes with key values 1, 2, 3, 4, and 5 are inserted, in the order listed, into an initially empty binary search tree. The layout of the resulting tree would resemble a linearly linked list. As you all know, item insertion, deletion, and retrieval operations in a linearly linked list all carry an undesirable, worst-case time complexity of $O(n)$. Fortunately, years ago, two mathematicians named Georgy Adelson-Velsky and Evgenii Landis designed specialized insertion and deletion functions for the binary search tree to ensure that the data structure always maintains its balance and thus avoids the linear time complexities plaguing the unbalanced trees.

Your task for this project is to implement the aptly named AVL tree, which requires both Adelson-Velsky and Evgenii's specialized insertion and deletion functions. To determine whether or not your code is working properly, you will write a function that prints the heights of the binary nodes contained in your AVL tree. You may find it helpful to compare your results with those obtained from an interactive, AVL tree visualization demo found at `https://www.cs.usfca.edu/~galles/visualization/AVLtree.html`.

Along with this handout, we also provided you with starter code for a standard binary search tree. You need to modify the included insertion function and implement the AVL deletion and height printing functions. Keep in mind that the associated AVL algorithms will require you to add helper functions to the `AVLTree` class and possibly a few more variables to the `BinaryNode` struct. With that said, you are free to scrap the provided code and write everything from scratch.

Input

Input commands are read from the keyboard. Your program must continue to check for input until the quit command is issued. The accepted input commands are listed as follows:

`i k` : Insert node with key value k into AVL tree.

`r k` : Remove node with key value k from AVL tree.
(When removing a node with two children, look to the InOrder predecessor in left subtree for the swapping node)

`h` : Print the height of each node using an inorder traversal.

`p` : Print the key value of each node using an inorder traversal.

`q` : Quit the program.

Output

Print the results of the `p` and `h` commands on one line using space as a delimiter. If the tree is empty when issuing the commands `p` or `h`, output the message `Empty`. If an attempt is made to remove a node not present in the tree, print `No node`. If an attempt is made to insert a node with a duplicate key value, print `Duplicate` without adding the new node to the tree. Do not worry about verifying the input format.

Sample Test Case

Use input redirection to redirect commands written in a file to the standard input, e.g. `$ ./a.out < input1.dat`.

Input 1

```
i 100
i 200
i 300
h
p
q
```

Output 1

```
1 2 1
100 200 300
```

Turn In

Submit your source code through blackboard. If you have multiple files, package them into a zip file.

Grading

Total: 100 pts.

- **10**/100 - Code style, commenting, general readability.
- **05**/100 - Compiles.
- **05**/100 - Follows provided input and output format.
- **80**/100 - Successful implementation of the AVL tree.