`Due: Nov. 27, 2023 at 11:59pm`

Overview

Building on your knowledge of Binary Search Trees, this lab will focus on implementing a Red-Black Tree.

Data Structures

A BST is a Red-Black tree if it satisfies the following criteria:

- Every node is either red or black.
- Every leaf counts as black.
- If a node is red, then both of its children are black.
- Every simple path from a node to a descendant lead contains the same number of black nodes.
- The root node is always black.

You should fill out all of the methods in the provided skeleton code `rb_tree.py`. You may add additional methods but should not add any private variables to the class. Like before, you should not alter any names for any of the classes, methods, or files. Your code will be graded using an autograder on Gradescope. To start you off, some test cases have been provided in `test_lab4.py`. You may not use any data structures from the Python standard library. Some inbuilt functions in python can be used as required.

Feel free to extend the implementation you have done for Lab3 and extend it for Lab 4 (However, if you would like to start fresh, please ask Paul (`ysoh@uoregon.edu`) for starter code, that has bst insert and delete implemented). As the textbook discusses in further detail, the insert and delete methods for red-black trees are an extension to bst insert and delete from Lab 3. You simply need to extend the functionality to support balanced insert and delete operations.

RB Tree

`print_with_colors(self, curr_node)` : Same as the `print_tree()` method provided in the skeleton code except it prints the color code along with the node. 'B' for black, 'R' for red. It also prints in preorder traversal format just like `print_tree()`.

`left_rotate(self, curr_node)` :

`right_rotate(self, curr_node)` :

`insert(self, data)` : Perform BST insert. Then check if the tree violates any of the Red-Black Tree criteria. If the criteria are violated, perform the appropriate rotations and or re-colorings. This can be done by implementing the `__rb_insert_fixup()` method.

`delete(self, data)` : Perform BST delete. Then check if any of the RBT criteria are violated. If so, perform the appropriate rotations and/or re-colorings. This can be done by implementing the `_rb_delete_fixup()` method.

Testing

Some sample test cases have been provided in `test_lab4.py` but this provided list is not exhaustive. You should provide tests for the following:

Test left rotation of intermediate node

- Insert the following elements using `bst_insert()`: $[7, 5, 9, 3, 6, 8, 10, 1, 2]$
- Left rotate node 9
- Test if the rotation was performed correctly.

Test colors after insertion

- Insert the following elements using `bst_insert()`: $[7, 5, 9, 3, 6, 8, 10, 1, 2]$
- Test the color of every node after the above insertion.

Add 3 of additional test cases of your own making.

It is strongly encouraged that you write even more tests.

Submission

Compress the `test_rb_tree.py` and `test_lab4.py` files and upload in Gradescope similar to the previous programming assignments. The test cases file should contain all the additional test cases that you have written to test your code.

Grading

The assignment will be graded as follows:

— AutoGrader - [35pts]

— Style - [5pts]

— Additional Test Cases - [7.5pts]

— DocStrings - [2.5pts]