CISC 233 Essential Algorithms Lab #4

# Lab Description

Perform a series of empirical tests on two different self-balancing search tree implementations, counting the number of comparisons and rotations used when adding elements.

## Notes

* NOTE: for this and all future Python labs in this class, ***your code must be written for Python 3.9***. There were some significant changes between Python 2.x and Python 3.x, and code written for one version may not work for another. See Python Version Help in Canvas for more information.

# Introduction

* Create a Python source file named of yourlastname.py replacing *yourlastname* with your actual last name. Include the following comment lines at the beginning of the file.

# CISC 233 LAB 4 Self-Balancing Binary Search Trees

# Author: <yourname>

# Date Created: <file creation date>

# Last Modified: <file modification date>

# Task 1 Red-Black Tree setup

* Find an implementation of RedBlackTree in Python (version 3 or higher)*.* Include the source information in a comment in your Python Source file. Note: your code must run in Python 3, so either find a version written for Python 3, or make necessary modifications to the code that you find.
* Add or modify driver code to test the red-black tree implementation with three datasets: once inserting elements *in order*, once in *reverse order*, and once in *random order*. Each of the three should add at least 32 distinct elements to the RedBlackTree. Do not add duplicate elements – each element added should be distinct.
* After adding the 32 elements, print out the tree to show its structure, verifying that the resulting trees are relatively balanced. The printout should show the structure of the tree, the values it contains, and the color (red/black) of each node, so that you can verify that it is in fact red-black correct at each stage.
* Add a second set of tests with three additional datasets: again, one with elements *in order*, once in *reverse order*, and once in *random order*. Each of the three should add at least 256 distinct elements to the RedBlackTree. Do not add duplicate elements – each element added should be distinct.
* After adding the 256 elements, perform a test to demonstrate that the elements in the resulting RedBlackTree are indeed in-order. Note: the trees will likely be too big to print out, so instead just test to ensure that the elements are in order. For example, use an in-order traversal of the tree and verify that the elements are returned in order.

# Task 2: AVL Tree Setup

(Note: Task 2 mirrors Task 1, except for AVL Tree rather than Red-Black Tree)

* Find an implementation of AVL Tree in Python (version 3 or higher)*.* Include the source information in a comment in your Python Source file. Note: your code must run in Python 3, so either find a version written for Python 3, or make necessary modifications to the code that you find.
* Add or modify driver code to test the AVL tree implementation with three datasets: once inserting elements *in order*, once in *reverse order*, and once in *random order*. Each of the three should add at least 32 distinct elements to the AVLTree. Do not add duplicate elements – each element added should be distinct.
* After adding the 32 elements, print out the tree to show its structure, verifying that the resulting trees are relatively balanced.
* Add a second set of tests with three additional datasets: again, one with elements *in order*, once in *reverse order*, and once in *random order*. Each of the three should add at least 256 distinct elements to the AVLTree. Do not add duplicate elements – each element added should be distinct.
* After adding the 256 elements, perform a test to demonstrate that the elements in the resulting AVLTree are indeed in-order. Note: the trees will likely be too big to print out, so instead just test to ensure that the elements are in order. For example, use an in-order traversal of the tree and verify that the elements are returned in order.

# Task 3 Empirical Testing

* Add code to your Red-Black Tree implementation to count the total number of rotations performed for each of the three insertion orders (in-order, reverse-order, and random-order) for datasets of size 256, 1024, and 2048.
* Add code to your AVL Tree implementation to count the total number of rotations performed for each of the three insertion orders (in-order, reverse-order, and random-order) for datasets of size 256, 1024, and 2048. (Use the same random-order data for both the Red-Black Tree and the AVL Tree)
* Your program should print out the results, indicating each of the following:
  + Balancing algorithm (Red-Black or AVL)
  + Data Size (256, 1024, or 2048 elements)
  + Insertion order (in-order, reverse-order, random order)
  + Number of rotations performed