Assignment #3, Part #2 of 2
AVL Tree Implementation

Instructor: Homeyra Pourmohammadali
BME 122 - Data Structures and Algorithms
Winter 2022
University of Waterloo

Due: 5:00 PM, Friday, March 18, 2021

Purpose of this assignment

In this assignment, you will practice your knowledge about **tree** by implementing a data type called **AVL tree**. The header file `avl-tree.h`, which is explained below, provides the structure of the `AVLTree` class with declarations of member functions.

Note that the `AVLTree` class is an extension through inheritance of your `BinarySearchTree` class. If you are not familiar with the concept of inheritance, e.g., how to define and use inherited variables and functions, read a simple tutorial here: `https://msdn.microsoft.com/en-us/library/84eaw35x.aspx`.

Instruction

Sign in to GitLab and verify that you have a project set up for your Assignment 3 (A3) at `https://git.uwaterloo.ca/bme122-1221/a3/WATIAM_ID` with the following files.

```
YOUR-WATIAM-ID
├── CMakeLists.txt
├── README.md
├── binary-search-tree.cpp
├── binary-search-tree.h
├── avl-tree.cpp
├── avl-tree.h
├── test.cpp
├── a3-1.pdf
└── a3-2.pdf
```

For this part of assignment, you need to modify `avl-tree.cpp` and maybe also `avl-tree.h` if necessary. You can design your own test case and code in `test.cpp`. It is optional and we will not grade this file.

You can use the same procedures in Assignment 0 to pull, edit, build, commit, and push your repo.

Description

You need to implement the new AVL versions of `insert` and `remove` operations that keep the tree balanced. Place your code into `avl-tree.cpp`. The pseudocode for them has been introduced in the lectures. When defining the AVL version of them, you may want to call the Binary Search Tree version of `insert` and `remove` inherited from your Binary Search Tree class, using `BinarySearchTree::insert` or `BinarySearchTree::remove`, to insert or remove the node from the tree, respectively. After that, you need to ensure that the tree is kept balanced by applying the appropriate AVL tree rotations.

Do not modify the signatures of the `insert` and `remove` functions to ensure that the test cases pass. The AVL tree should be kept balanced after calling just insert or remove, so any re-balancing operations should happen within these functions rather than outside of them. You may create additional functions and/or attributes in the .h and .cpp files to help complete the tasks, if needed.

Brief Notes:
Assume that all rotation functions do not need to check the balance of the final tree.
Recall from lectures that an inbalanced node is a node whose subtrees are not balanced. i.e. a node for which (`HrHl`) is greater than 1, where `Hr` and `Hl` represent the depths of the left and right subtrees for this given node.

For all rotation functions:

The `parentNode` parameter is the parent of the node that is imbalanced. In the case that the root of the tree is unbalanced, this node is `NULL` (since the root does not have a parent). The `isLeftChild` parameter is true when the inbalanced node is the left child of the parent node; else this value is false. This function returns true for a successful left rotation.

Member functions:
`depthBelow(Node *n)`: Computes the maximum depth of the tree from the root node `n`. This function follows the convention that a single node has depth equals to 1. Thus, if the root node is a leaf node, the expected output would be 1.

`singleLeftRotation(Node *parentNode, bool isLeftChild)`: Carries out a single left rotation on the subtree. Returns `true` on success. Assume that no balance-checking needs to be carried out.

`singleRightRotation(Node *parentNode, bool isLeftChild)`: Carries out a single right rotation on the subtree. Returns `true` on success. Assume that no balance-checking needs to be carried out.

`leftRightRotation(Node *parentNode, bool isLeftChild)`: Carries out a left-right rotation on the subtree. Returns `true` on success. Assume that no balance-checking needs to be carried out.

`rightLeftRotation(Node *parentNode, bool isLeftChild)`: Carries out a right-left rotation on the subtree. Returns `true` on success. Assume that no balance-checking needs to be carried out.

`pathToNodeStack(DataType val)`: Computes a stack that contains all nodes that are traversed in a tree, in order to reach the node with the given value. The last inserted element of the returned stack should be the parent of the node that you are trying to reach. You can assume that this function will always be called on a valid `val`.

`updateNodeBalance(std::stack<BinarySearchTree::Node*> *pathToNode, DataType val)`: A generic function that is called by every insert and delete operation to rebalance the tree. This function determines which of the rotation functions to call. The `pathToNode` parameter is the output of the `pathToNodeStack` function and the `val` is the value of the node that was either inserted or deleted. This function returns `true` if the balancing was successful.

`insert(DataType val)`: Inserts a node with value equals to `val` into the avl tree. Returns the output of `updateNodeBalance`.

`remove(DataType val)`: Removes a node with value equals to `val` into the avl tree. Returns the output of `updateNodeBalance`.

Marking

We will try different inputs and check your output. We will only test your program with syntactically and semantically correct inputs.

Part 2 counts 40% of Assignment 3, which is 40 points in total.

Your program runs and does not crash during the test: + 10
Passes Test Cases: + 5 each, in total of 30