Lab Exercise #4
Due Date: Mar 10th 2024
Binary Search Trees

Description:
For this programming assignment, you will implement a **Binary Search Tree.** A Binary Search Tree (BST) is a data structure where each node has at most two child nodes, usually referred to as the left child and the right child. The key property of a BST is that for each node:

- All nodes in its left subtree have values less than the node's value.
- All nodes in its right subtree have values greater than the node's value.

starter.zip (Starter Code):
The following is a brief description of the starter code provided to you:

- **bst.h**
    - This header file declares a BST_Node class and a BST class
        - **BST_Node** class: The BST_Node class contains the following public member variables:
            - **Type key**: this refers to the key stored in the node of which the node is identified by
            - **BST_Node* left**: The pointer to the left child of the node
            - **BST_Node* right**: The pointer to the right child of the node
        - **BST class**: The BST class contains a single public member variable: **BST_Node* root**, which will store the pointer to the root node of the BST (Note that the root is normally private but we have made it public for testing purposes). The following are the public functions of the BST:
            - **Copy Constructor**
                - The **copyTree** function is a private helper function that will be used to implement the copy constructor. You can use the copyTree function recursively to deep copy the BST. You will have to make a call to this function using the other BST's root and recursively work your way down the tree to copy each node.
            - **Destructor**
                - You will use the **clearTree** private function to recursively delete all nodes of the tree starting from the root.
            - **Copy Assignment**
                - The **copyTree** and **clearTree** private helper functions will be used to implement the copy assignment.
            - **insert**
                - The insert function will take a parameter "key" that will have to be inserted into the BST. You can use the private **insertRecursive** helper

function to insert the key in a recursive manner.
    - You should not insert the key if it already exists in the binary search tree
- **deleteNode**
    - The deleteNode function will take a parameter "key" that will have to be deleted from the BST. You can use the private **deleteRecursive** helper function to delete the node containing the key in a recursive manner.
    - The private **minValue** helper function will be of assistance when implementing the logic to delete the key. The minValue function will find the minimum value in the subtree rooted at the node (including the node itself) taken in as a parameter.
    - You can leave the tree undisturbed if the key to be deleted does not exist in the tree.
- **find**
    - The find function will return true if a particular key taken in as a parameter exists within the tree and false if it does not exist. You can use the private **findHelper** function to recursively search for a key starting at the root.
- **printTreeInOrder**
    - The printTreeInOrder function will return a string containing the keys as a result of traversing the tree in-order. The private **printTreeInOrderHelper** function can be used to recursively traverse the tree inorder.
    - Notice that the printTreeInOrder helper function takes in a string (passed by reference) as a parameter. You can simply add the key to the string and add a space after adding the key
        - For example you can do: **s += to_string(key) + " "** during the inorder traversal.
    - An example format (of a BST containing the nodes with keys 1, 2, 3, and 4) of the returned string should be as follows:
        - "1 2 3 4 "

Contract (TASK):
Your task is to implement the methods of the BST class. **You have been provided with a few functional function definitions of some top level methods (insert, deleteNode, printTreeInOrder, find etc). You must implement all of the remaining methods for which you have not been provided functional**

definitions (the helper functions, copy assignment, copy constructor, destructor etc.). Do not create additional cpp files to define these functions, make sure that they are part of the header file.

Deliverables and Submissions:

In a zip folder named using the format **<FirstName>-<LastName>-<UIN>-<LE4>.zip** you must submit the following file(s) to Canvas:

- **bst.h** (containing the functional method definitions of the BST class)

Do not use angular brackets in the name of the submission folder.

Grading:
Coding (20 points)
Each method of the BST will be tested using an automatic testing infrastructure. You will be provided with the testing infrastructure which you can use to evaluate your implementations.

The grading rubric is as follows:

- Each of the following methods will be tested out for each of the three classes:
  - Copy Constructor: 2 points
  - Copy Assignment: 2 points
  - insert: 5 points
  - find: 2 points
  - deleteNode: 7 points
  - printTreeInOrder: 2 points

Note that many of these functions have dependencies on each other (for example, to pass insert, you must pass find first).
In case of any memory leaks in your program, we will subtract 2 points.

testInfrastructure.zip (Test Methodology):
We are providing you with an automated test infrastructure that can be used to check the correctness of your code. The testing infrastructure is constructed as follows:

- **bst_test.cpp**: This cpp file defines all the test functions that have been mentioned in the grading rubric above. It also runs these tests and outputs your final score.
- **test.py**: This file compiles bst_test.cpp in order to produce the executable. It then runs this executable which will indicate your score (as well as the test cases you have passed or failed). It will also run valgrind on the executable file to indicate the presence of memory leaks if they exist.

Note: There is a possibility that one or more of your methods causes a segmentation fault which in turn causes the test script to crash when calling that particular method. In such a case, we will not be able to calculate your score for the assignment and you will be assigned a default grade of 0. After the grades are posted, you will be given 1 week to rectify your error(s) and resubmit the assignment. You will be subject to a 25%

penalty on the grade obtained from the resulting resubmission of the assignment.

How to test your program natively (run your own test cases):
You can use main.cpp provided in the starter.zip folder to instantiate BST objects. You can then test the functionality of your methods by calling these methods and checking whether what they return is what you expect. To compile your own tests in main.cpp, **you can use the makefile provided within starter.zip to compile your program.** In your terminal:

- Run **make** on the terminal (this will create an executable named main)
- Next, run **./main** on your terminal to run your native tests
- Finally run **valgrind --leak-check=full ./main** to check whether there are memory leaks.

How to use testInfrastructure:
To check your score using testInfrastructure, you can follow either of the two options mentioned below:

- **Option 1:** Move your bst.h into the testInfrastructure directory. After this, run **python3 test.py**
- **Option 2:** Move your bst.h into the testInfrastructure directory. **Ensure that the makefile in this directory is the one given to you originally in the testInfrastructure.zip folder and not in the starter.zip folder.** In your terminal:
    - Run **make** in order to compile the program (which will compile bst_test.cpp and create an executable named bst_test).
    - Next, run **./bst_test** on your terminal to actually run the tests and output your score.
    - Finally run **valgrind --leak-check=full ./bst_test** to check whether there are memory leaks.

The current starter code provided to you will have the memory leaks if tested out on bst_test.cpp. Once you implement functional code, these memory leaks will disappear.

Please also make sure that you run the test script a few times (3 - 4 times should be sufficient) to ensure that your code is consistently passing test cases in different random runs of the test script.

In bst_test.cpp, you can set the DEBUG FLAG to 1 which will allow you to print the tree in level order when the level order function is called within bst_test.cpp. You can reset it back to 0, to prevent any output from being printed out. You are recommended to use small sizes to test your binary tree first before you attempt to run the test script (which uses larger sizes); you can also configure the testing script to test smaller sizes (but please make sure you also test your code with the sizes initially given in the test script).