COMP 215 Algorithms

Lab #8

In this lab, we will review Binary Search Trees.

There is nothing to hand in, but much of this code will appear in a slightly modified form when you have to implement a Red-Black Tree for your project. The main difference is that the value component of a red-black tree node for your project will be a record instead of an int, and the red-black tree node will inherit from the container class.

First, as in the queue, we program a class for a basic node in the tree.

1. Program a class `treeNode` that contains one public members: an integer called `value`, and three protected members, a pointer to a treeNode called `left`, a pointer to a treeNode called `right` and a pointer to a treeNode called `parent`.
2. Program a constructor for this class that takes an integer, and initializes `value` to this integer, and initialises `left` and `right` to NULL.

We're now ready to program the tree itself. The class that implements the actual tree will need to play with the pointer elements of the treeNode, so something needs to be added to the treeNode class so that you can access these elements from inside the tree class.

3. Program a class `binarySearchTree` that contains a private pointer to a node called `root`.
4. Program a (public) constructor for the tree that takes no input and only initializes the `root` to NULL.
5. Program a (public) destructor for the tree that only calls the function `destroy_tree` on the root of the tree. The function `destroy_tree` is a helper function we implement below.
6. Program a *private* function `destroy_tree` that takes a pointer to a treeNode called `leaf` as input, and, if that pointer is not NULL, calls `destroy_tree` on `leaf−>left` and on `leaf−>right`, and then deletes the node `leaf`.
7. Program a *public* function `insert` that takes an integer `key` as input and does not return an output. If `root` is NULL, then the function should assign to `root` a new node that has `key` as its value, and NULL as its `left`, `right` and `parent` pointers. Otherwise, the function should call the private helper function `insert_help` on input `key` and `root`.

8. Program a *private* helper function `insert_help` which takes an integer `key` and a pointer to a treeNode `leaf` as its input, and does not have an output. This function will assume that the pointer `leaf` is not NULL.
   If `key` is less than `leaf−>value` and `leaf−>left` is not NULL, recursively call `insert_help` on input `key` and `leaf−>left`.
   If `key` is less than `leaf−>value` and `leaf−>left` is NULL, assign to `leaf−>left` a new treeNode that has `key` as its value and NULL as its `left` and `right` pointers. Be careful to set the value of the `parent` pointer of the new treeNode correctly.
   If `key` is greater than `leaf−>value` and `leaf−>right` is not NULL, recursively call `insert_help` on input `key` and `leaf−>right`.
   If `key` is greater than `leaf−>value` and `leaf−>right` is NULL, assign to `leaf−>right` a new treeNode that has `key` as its value and NULL as its `left` and `right` pointers. Be careful to set the value of the `parent` pointer of the new treeNode correctly.
   If `key` is equal to `leaf−>value`, stop and do nothing (we do not insert multiple copies of a value in the tree).

9. Program a *public* function `search` that takes an integer `key` and returns a pointer to a treeNode. It only runs the private helper function `search_help` on input `key` and `root`.

10. Program a *private* helper function `search_help` that takes an integer `key` and a pointer to a treeNode `leaf` as input, and returns a pointer to a treeNode.
    If `leaf` is NULL, return `false`.
    Else if `leaf−>value` is equal to `key`, return true.
    Else if `key` is less than `leaf−>value`, recursively call `search_help` on input key and `leaf−>left` and return its result.
    Else recursively call `search_help` on input key and `leaf−>right` and return its result.

11. Program a *public* function `remove` that takes an integer `key` and does not return an output. This function is a little complicated and I will review it during the lab.

12. Program a *public* function `inOrderTraversal` that neither takes, nor return an input (it will only print stuff on the screen). This function only calls the private helper function `inOrder_help` on input `root`.

13. Program a *private* function `inOrder_help` that takes a treeNode pointer `leaf` as input and does not return an output.
    If `leaf` is NULL, the function stops without doing anything.
    If `leaf` is not NULL, then the function first calls `inOrder_help` on `leaf−>left`, then prints `leaf−>value` on the screen, then calls `inOrder_help` on `leaf−>right`.

That's it! Now let's run this code on a few tests.

14. Write a main function that creates an empty binary search tree, and then prompts the user to enter positive integers. The main function should prompt for new integers an enter each integer in the binary search tree using its `insert` function, until the user enters 0 or a

negative integer, at which points it stops prompting (and the 0 or negative integer should not be inserted in the tree). Try to enter numbers in what seems like a random order to you.

15. The main function should then prompt the user for positive integers to search in the tree, and for each integer it gets, use the `search` function of the tree to determine if the integer was inserted in the tree. It should then print the result of the function call on the screen. Keep doing this until the user enters 0 or a negative integer.

16. The main function should then run the function `inOrderTraversal`. Do you notice anything special about this output?