Data Structures (2028C) -- Spring 2025 – Lab 9

Topics covered: Working with Binary Search Trees

Lab due: ***Sunday, Apr 6 at 11:55PM for Monday Section***
Tuesday, Apr 8 at 11:55 PM for Wednesday Section

Objective:

The objective of this lab is to implement and balance a Binary Search Tree.

Scenario:

In your first homework assignment, you created a program that counted the number of words and the frequency of each letter. This assignment will leverage your code from that assignment to read a file but with a modification. Instead of counting the frequency of each letter, you are going to count the frequency of each word. You will use a Binary Search Tree to store each word along with its count. This needs to be written using C++.

Requirements:

1. Create a templated Binary Search Tree class with the following methods. You may need or find it easier to create private methods and variables to simplify or eliminate duplicate code in the methods below such as **RotateRight**, **RotateLeft** or **NodeHeight**.
    a. **Constructor**
    b. **Destructor**
    c. **Insert** – accepts a value, creates a node, and inserts the node into the tree in the appropriate location. This should rebalance the tree as necessary, so the tree remains balanced at all times. If the value already exists in the tree, this should throw an error.
    d. **Find** – accepts a value, locates the value in the tree and returns a pointer to the node. If the value isn't in the tree, it will return a null pointer.
    e. **Size** – returns the number of elements in the tree as an integer.
    f. **GetAllAscending** – returns an array with each node stored in order from smallest to largest (based on the sorting value, not the other data in the node).
    g. **GetAllDescending** – returns an array with each node stored in order from largest to smallest (based on the sorting value, not the other data in the node).
    h. **EmptyTree** – removes all nodes in the tree in a way to avoid memory leaks.
    i. **Remove** – accepts a value from the user, finds the value, and removes it from the tree. A pointer to the removed node is returned. This should rebalance the tree as necessary, so the tree remains balanced at all times.

NOTE: PLEASE MAKE SURE THAT YOU PRINT THE HEIGHT AND BALANCE FACTOR OF EACH NODE FOR OPTION 1(F) AND 1(G)

NOTE: USE RECURSIVE FUNCTIONS FOR Insert, Find, Size, GetAllAscending, GetAllDescending, EmptyTree, and Remove Method.

2. Modify your program from Homework 1 to read the input file and count the frequency of each word using the Binary Search Tree in step 1.

   This is likely not the best data structure for performing this function but this is the data structure we are currently learning about so we will use it.

   Your program should create a class made up of a string and an int that will be used as the type of the templated Binary Search Tree i.e. the data to be stored in the tree. **The string will be the word you are reading in and should be the key you are comparing at each level**. The int is the count for that word. You may need to overload <, > and/or == operators for this class to be used in your Tree. You can use the **Find** function to find the word in the tree then update the count via the returned node pointer. If **find** returns a null pointer, you will need to call **Insert**. Once the file is read and loaded to the tree, let the user search for a word. If it is found, return the count. If it is not, display a message. Also, allow the user to see a list of all words with count in alphabetic order both ascending and descending (Task 1 part f and g) ALONG WITH THE HEIGHT AND BALANCE FACTOR OF THE NODE, provide an option to view the size, empty the tree and remove a word. Discuss in your lab report the performance difference you would expect without vs. with balancing. This includes both the insert/remove performance and finding performance.

Submission:
Submit all source code files and any required data files in a zip file. Include a write up as a PDF including:

- The name of all group members
- Instructions for compiling and running the program including any files or folders that must exist.
- What each group member contributed. If the contributions are not equitable, what portion of the grade each group member should receive.

Submission should be submitted via Canvas.

Grading:

1. 20% - Lab attendance
2. 15% - BinarySearchTree class functions Insert, Find and Remove work correctly (except for keeping the tree balanced) including avoiding memory leaks.
3. 15% - BinarySearchTree stays balanced at all times with a difference of the longest path to a leaf and the shortest path to a leaf no greater than 2.
4. 15% - The other required functions of the BinarySearchTree work correctly including avoiding memory leaks.
5. 10% - Your main code correctly reads the input files and creates a Binary Search Tree.
6. 10% - Your main code correctly Allows the user to search for words and returns the correct result and outputs the full list in ascending and descending order.
7. 15% - Lab report contains all required information and is well written.

If the program fails to compile, the grade will be limited to a max grade of 50%.