CS 2028 C 002
LAB 9 Report

Kyle Russell, russek5

11/07/2023

Overview:

The purpose of this lab is to create and implement a binary search tree in C++. Additionally, the tree must be an AVL or similar style tree. An AVL tree is a tree that is sorted by value, and its order is balanced using rotations so that the difference between the longest path to a leaf node and the smallest is no greater than 2.

Insertion, Deletion and Searching

Insertion into the tree is handled recursively. It follows the standard binary search tree instructions of left child being less than the node, and right child being greater in each subsequent node. The difference in an AVL tree is that the root node is somewhere in the middle of the data, the farthest left is the lowest and the farthest right is the greatest.

In the balancing of the tree, a balance factor is calculated. It is simply difference in the height of the left child vs the left child. The height of a particular node is calculated as the greater of its children's heights, plus one. These heights are kept track of and updated as the tree is rebalanced using the previous formula. See Figures 1 and 2 for an example of an insertion and rotation.

In terms of performance, the tree itself carries an O(n) time complexity in its worst case. The AVL tree also carries an O(n) time complexity, with the added O(1) for each element in the tree that needs to be rotated, meaning that the complexity becomes O(n) at the worst. This means that theoretically, a non-AVL tree would be expected to perform better in all use cases for insertion.

Deletion is very similar to insertion, with the added step of deleting the pointer and reattaching its children. Afterwards, the tree must still be rebalanced to account for any imbalances that result from the removal of the node.

Searching does not follow a recursive structure. Using the fact that the left is less and the right is greater, the search can be accomplished by using a while loop. Each iteration of the loop points the current node pointer to either the left or right child depending on the comparison of the item to the node's data. If the two items are equal, a node pointer is returned. If no equality can be found, the node pointer ever points to null, returning a null pointer.

Other functions

The displayOrderAscending function will loop recursively through the tree to the lowest value (the lowest and farthest left leaf node), and will push a pointer to the data to a vector of nodes that is passed by reference. This will create a list of each node in ascending order that can be easily printed to the console, thanks to the "ostream& operator" << function defined globally. The same applies to the descending variant of the function, except in reverse. Output of the functions can be found in figures 3 and 4.

Conclusion

Binary search trees are very complex data structures that can hold a lot of data at once. While maybe not being the most efficient means of storing data, they are simple to traverse and read. Binary search trees are used in the map and set libraries in most programming languages, are used in some machine learning algorithms, for heaps and for some compression algorithms.

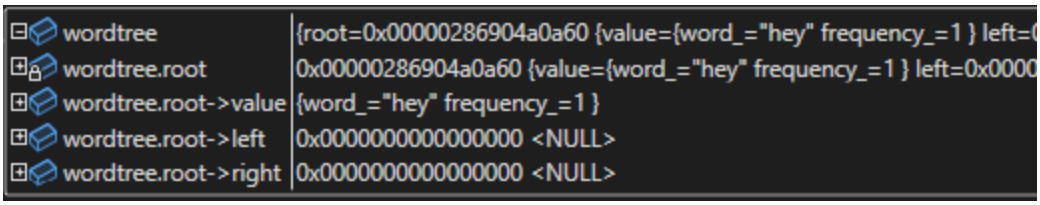

Figure 1: The word "hey" is inserted into the tree

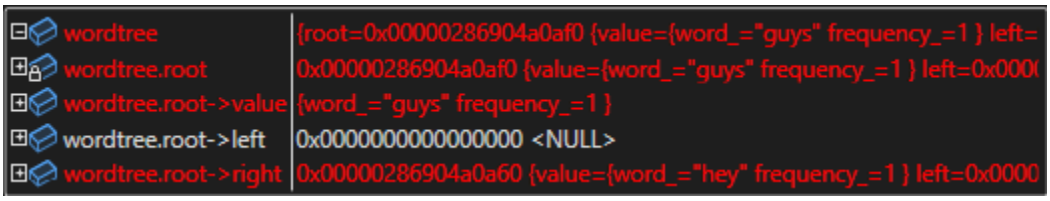

Figure 2: The word, "guys" is added, and a rotation is executed placing "guys" as the root.

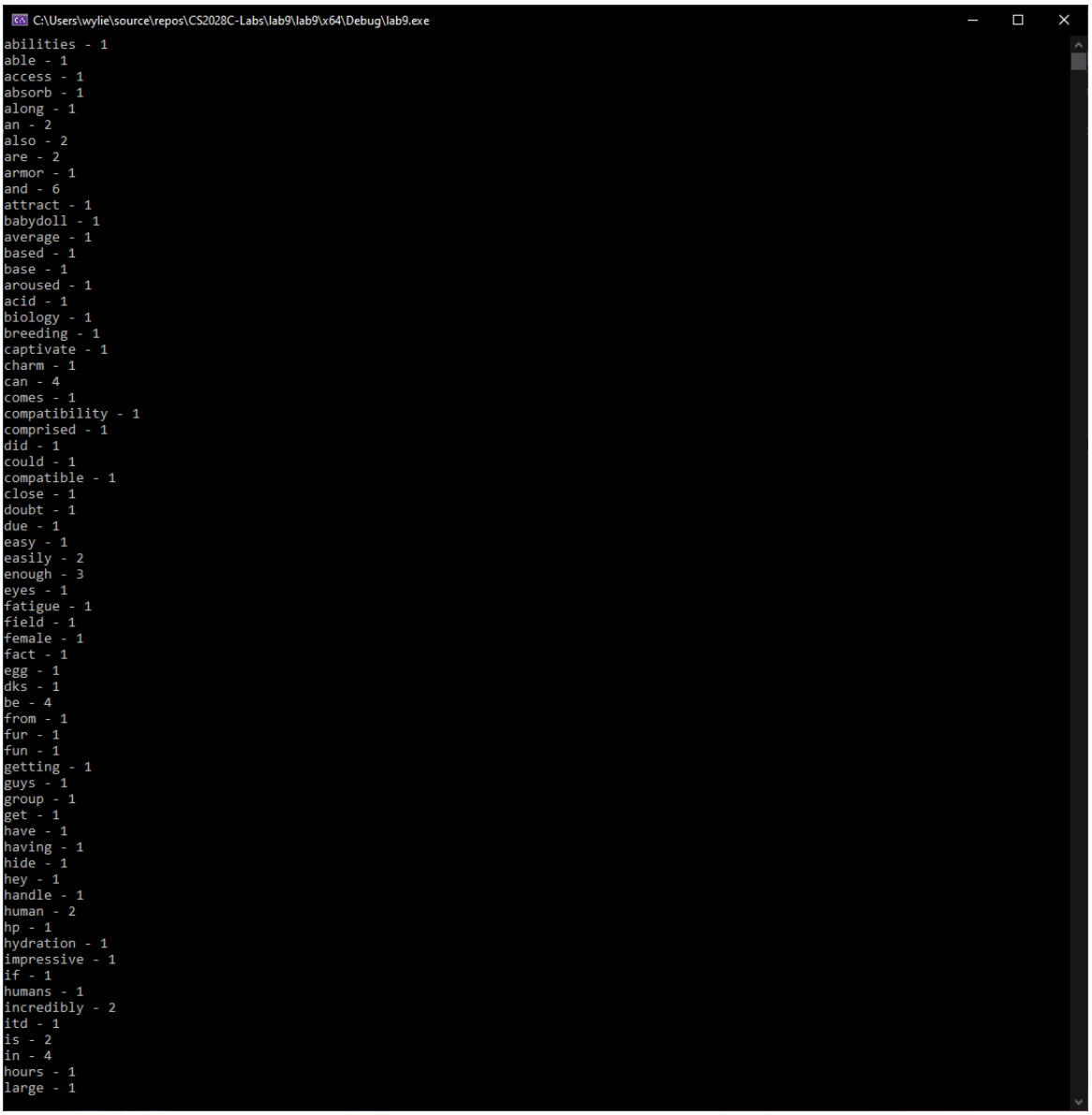

Figure 3: Please just look at the fact that these are in order

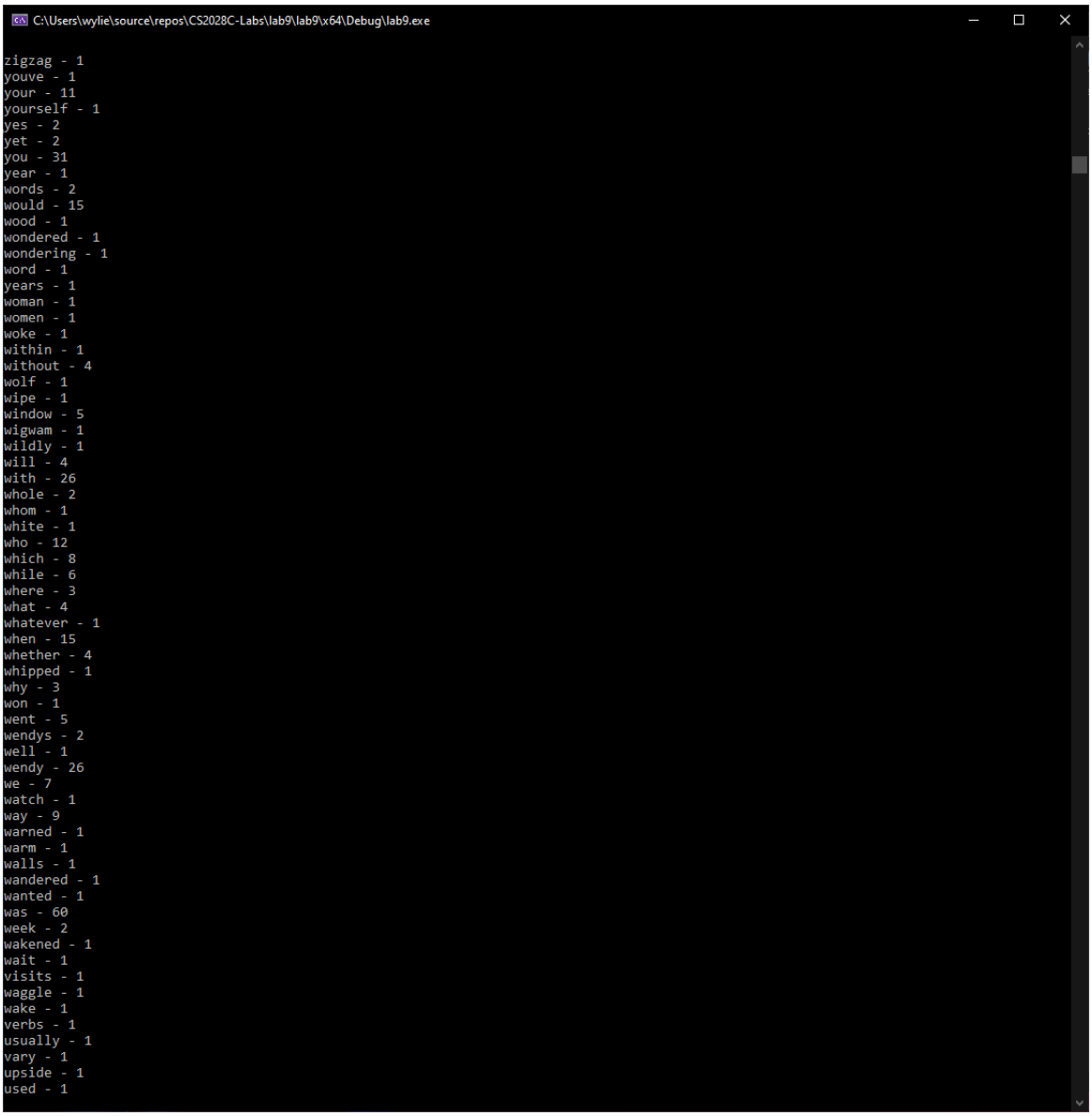

Figure 4: The elements (mostly) in reverse order.