Project 1 Report

Vinay Reddy Ratnam

Reflection:

Next time I create a program in c++ that takes in inputs from the command line, I will start using getline with cin much more than using "cin >>" because ">>" does not take the endlines at the end of the user inputs into account leading to a lot of complications when trying to read user input. I also want to use the regex library in c++ next time because it will take less time than manually checking the gator IDs and names.

When I started the project, I thought that it would be smart to take each node's level (with the root node starting at level 1) instead of their height as an int variable in each node class. After completing the project, I realized it is much smarter to keep each node's height than level as a variable because of the way recursion keeps the nodes deepest in the tree on the top of the stack. Additionally, next time I create an AVL tree, I think I can create a better deletion function that handles the edge case of deleting the root node with less lines of code and more efficiently.

Currently, a lot of the functions from the AVLTree class have to use private functions in the same class because the root is a private variable. Next time I create an AVLTree, if I am okay with managing memory outside the class, I could implement a rootGetter() function that would make it possible to access the tree from outside the class which would make it possible to create a lot of these public functions without the need for their private versions.

Worst Case Complexity Analysis:

insert NAME ID: O(m + log(n)) where m is the number of characters in the individual's name as checking if a name is valid requires you to check every character. n is the number of nodes in the tree and it takes in the worst case, O(log(n)) time to recursively move through the tree because it is a BST where every recursion cuts the amount of nodes that need to be searched by approximately half.

remove ID: O(log(n)) where n is the number of nodes in the tree as it takes in the worst case, O(log(n)) time to recursively move through the tree because it is a BST where every recursion cuts the amount of nodes that need to be searched by approximately half. Once you get to the desired node, removing the node takes constant time.

search ID: O(log(n)) where n is the number of nodes in the tree. You can use the inputed ID to recursively search through the tree with a binary search, giving you a time compelexity of O(log(n)).

search NAME: O(m + n) where m is the number of characters in the inputed name as checking if the name is valid requires you to check every character and n represents the number of nodes in the tree as you have to recursively search through every node to check if they have the inputed name.

printInorder: O(n) where n is the number of nodes as the recursive calls touch every node.

printPreorder: O(n) where n is the number of nodes as the recursive calls touch every node.

printPostorder: O(n) where n is the number of nodes as the recursive calls touch every node.

removeInorder N: O(m + n) where m is the number of digits in N because every digit is checked to check if N is a valid input and n is the number of nodes in the tree as every node is recursively called to in the removeInorder function.