**Project 1: Gator AVL Project Documentation**

**By Aarithi Rajendren**

Insert *NAME ID*

The worst-case time complexity of the insertNode function is O(m + log n) where n represents the number of nodes in the tree and m represents the number of letters/numbers in the name/ID wanting to be inserted. This is because the functions checkName, checkID, and removeQuotes all have a time complexity of O(m) where m is the length of the name or ID since the functions loop through each letter/number to make sure it is valid and for the removeQuotes function, the function loops through all the characters in the string to remove any quotes. Then the insertNode function recursively calls the insert function. The insert function’s time complexity is O(log n) where n represents the number of nodes in the tree because to find out where to insert the node the function cuts off half off the nodes that have IDs that are either smaller or greater than the ID that will be inserted. The balance function has a constant time complexity of O(1) since there is no iteration through the nodes in the tree and it just calls one of the rotation functions that also have a time complexity of O(1) because they perform the same number of operations regardless of the size of the tree. Thus, the overall insertNode function would have a time complexity of O(m + log n).

Remove *ID*

The worst-case time complexity of the removeID function is O(log n + h) where n represents the number of nodes in the tree and h represents the heigh of the tree. This is because the function calls a helper function called scan, which calls a function called searchForID. The searchForID function has worst-case time complexity of O(log n), where n represents the number of nodes in the tree, because the function will disregard half of the nodes that have IDs less than or greater than the ID that is going to be removed when searching for the node with a matching ID. Thus, the scan function also has a worst-case time complexity of O(log n) since it calls the searchForID function within it. The removeID function also calls a function called removing which is a recursive function. The function called removing has a worst-case time complexity of O(log n) since it will disregard half of the nodes depending on whether the IDs in the nodes are less than or greater than the ID that is being removed. Furthermore, a helper function called getInOrderSuccessor is also called and it has a time complexity of O(h) where h represents the height of the tree since in the worst case we would have to traverse the entire height of the tree to find the inorder successor. The balance function is also called and it has a time complexity of O(1) as already mentioned. Thus, the removing function has a worst-case time complexity of O(log n+ h). Therefore, the overarching removedID function has a worst-case time complexity of O(log n + h).

Search *ID/Name*

The worst-case time complexity of the search function is O(n + m) where n represents the number of nodes in the tree. This is because this function calls the checkName and checkID function which both have a worst-case time complexity of O(m) where n is the length of the name or ID since the functions loop through each letter/number to make sure it is valid. Also, the search function calls searchForName and searchForID. searchForName has time complexity of O(n) where n represents the number of nodes in the tree because in the worst case the function would have traverse through every node in the tree until finding the name. searchForID has a time complexity of O(log n) where n represents the number of nodes in the tree because the function will disregard half of the nodes that have IDs less than or greater than the ID that is going to be removed when searching for the node with a matching ID. Thus, the search function will have a time complexity of O(n + log n + m), however since n is a more dominant function than log n, the time complexity of the search function ends up being O(n + m).

PrintInOrder

The worst-case time complexity of the printInOrder function is O(n) where n is the number of nodes in the tree. This is because the function recursively calls a helper function called printIn, which has a worst-case time complexity of O(n) where n is the number of nodes in the tree. The function printIn traverses through each node in the tree, which is why its time complexity is O(n). The for loop in the printInOrder function has a time complexity of O(i) where i represents the number of elements in the vector which holds all the names stored in the nodes in the tree. Thus, printInOrder has a worst-case time complexity of O(n+i) but since n is a more dominant function than i, the final time complexity of printInOrder ends up being O(n).

PrintPreOrder

The worst-case time complexity of the printPreOrder function is O(n) where n is the number of nodes in the tree. This is because the function recursively calls a helper function called printPre, which has a worst-case time complexity of O(n) where n is the number of nodes in the tree. The function printPre traverses through each node in the tree, which is why its time complexity is O(n). The for loop in the printPre function has a time complexity of O(i) where i represents the number of elements in the vector which holds all the names stored in the nodes in the tree. Thus, printPreOrder has a worst-case time complexity of O(n+i) but since n is a more dominant function than i, the final time complexity of printPreOrder ends up being O(n).

PrintPostOrder

The worst-case time complexity of the printPostOrder function is O(n) where n is the number of nodes in the tree. This is because the function recursively calls a helper function called printPost, which has a worst-case time complexity of O(n) where n is the number of nodes in the tree. The function printPost traverses through each node in the tree, which is why its time complexity is O(n). The for loop in the printPost function has a time complexity of O(i) where i represents the number of elements in the vector which holds all the names stored in the nodes in the tree. Thus, printPostOrder has a worst-case time complexity of O(n+i) but since n is a more dominant function than i, the final time complexity of printPostOrder ends up being O(n).

PrintLevelCount

The worst-case time complexity of the printLevelCount function is O(n) where n represents the number of nodes in the tree. This is because the function calls a recursive helper function called getLevelCount which has worst-case time complexity of O(n) where n represents the number of nodes in the tree. The function getLevelCount traverses through each node in the tree to check the level of the tree so its time complexity would be of O(n). Thus, the overarching time worst-case time complexity of printLevelCount would be O(n).

RemoveInOrder *N*

The worst-case time complexity of removeInOrder is O(n) where n represents the number of nodes in the tree. This is because the function calls a helper recursive function called removeNth whose worst-case time complexity is O(n) where represents the number of nodes in the tree since in the worst case the function would have to check every node to find the node that has the target ID. Furthermore, the balance function has a time complexity of O(1) as previously mentioned. So, the worst-case time complexity of removeInOrder is O(n).

Reflection

From this assignment, I got a deeper understanding of trees and how to traverse them using recursion. I have never been very comfortable with recursion since its hard for me to envision the final result after the recursion has run its course. However, I realized how efficient recursion is, so I wanted to implement that in this project. I made sure to draw a tree for every function and try to visualize how the tree looks after each insertion, deletion, rotation, etc. This assignment made me learn how important it is to visualize what I am trying to do and make sure that my code accurately reflects my intentions.

The one thing I would do differently if I had to start over would be to create my leftRightRotation and rightLeftRotation functions earlier on. I wanted to try the LR and RL rotations with just two functions (leftRotation and rightRotation), however, this proved to be very difficult for me. Especially, when trying to write test cases for the LR and RL rotation I kept getting stuck since the tree was getting balanced properly and I couldn’t see exactly what was happening with the nodes in the tree so I spent a lot of time trying to debug and figure out what is happening to the tree at each step. I think it would’ve saved me a lot of debugging time if I had just created separate LR and RL functions at the beginning.