COP3530 Project 1 AVL Tree Report: Neil Patel

Disclaimer: In the functions insert NAME ID, remove ID, search ID, search NAME, and remove In-order N, there are string comparisons being made by either checking/ comparing the names or the IDs. I am adding this disclaimer here because otherwise, it would make the report too long to read as my program uses the comparison of strings extensively and writing about it here to encapsulate those functions mentioned above will make the report better.

Comparing strings: If string names are being compared/ checked with other string names, then the worst-case time complexity of this operation is O(1) and O(1) space. I am saying O(1) time because with tons of research, it is safe to assume that the average length of a person's full name is about 13 characters, with the longest name ever recorded to be 747 characters long. It is safe to assume that a human cannot possibly have a name longer than 747 characters. Since O(13) and O(747) are constant numbers, asymptotic analysis says that we can drop known constants. Making the worst cast time complexity of comparting string names O(1) and O(1) space. *The overall impact this will have on the program when comparing/ checking string names will be nothing because it is a O(1) time and space operation.*

Comparing IDs: If string IDs are being compared/ checked with other string IDs, then the worst-case time complexity of this operation is O(8) -> O(1). This is because there are always 8 characters in the string ID that need to be compared. As 8 is a constant number, asymptotic analysis says that we can drop known constants. Making the worst cast time complexity of comparting string IDs O(1) and O(1) space. *The overall impact this will have on the program when comparing/ checking string IDs will be nothing because it is a O(1) time and space operation.*

The AVL class in general: In this class we have 9 operations that the AVL Tree data structure supports. Within the class, supports node construction where each node that is created has attributes such as name, ufid, left pointer, right pointer, and height. It is important to know that an AVL Tree Data Structure is a self-balancing BST, which makes most of the operations O(log n) time complex because the tree is sorted from lest to greatest and always balanced. It is important to note that most, if not all, of these 9 operations are using helper function to be called as it makes for a better design for function calling from the main.cpp file, as the AVL tree objects root attribute is not being used. Also important to note that the name and ID attributes for the nodes are stored as strings.

Insert NAME ID:

1.) The time complexity of this function is O(log n) in worst case and O(1) space complexity.
2.) Where "n" is the number of nodes (accounts) in the AVL Tree.
3.) This is because every time we insert a new node into the AVL Tree (with height 1), the UFID of that node gets compared with the root->left and root->right children, where every iteration inside the recursive call gets rid of half of the nodes at a time because the

tree is balanced. This leads to the node being inserted into the correct BST position. After inserting into the position, the function recursively checks the height of the inserted node, then the node above it, then the node above that one, and so on, while also calculating the balance factor (of the same path) of the newly calculated height node. If the balance factor for a node is 2 or -2 for any node, then the insert function will then check which way the imbalance is (left-left, left-right, right-right, right-left) and call either or the rotate functions (rotateRight or rotateLeft). The calculation of the new height of the node, the calculation of that node's balance factor, the comparison of the balance factors (2 and -2), calling the rotation functions, the rotations themselves, and the height calculations after rotations, are all O(1) constant time operations. Leading to a O(log n) insertion time with O(1) constant memory.

Remove ID:

1.) The time complexity of this function is O(log n) in worst case and O(1) space complexity.
2.) Where "n" is the number of nodes (accounts) in the AVL Tree
3.) This is because the remove ID function takes advantage of the already balanced tree data structure. When given the ID of a node to remove, we can recursively check the roots->left and roots->right children to see if the ID is greater than it or less than the child and recursively move down the path of the tree until it finds the specified ID. Each pass, the program removes half of the possible outcomes to eventually reach the node that needs deletion. Therefore, this is a O(log n) time operation. When the recursive traversal reaches the node with the specified ID, the node can be removed in 3 cases (node has no children, node has 1 child, node has 2 children). If the node has one or no children, then we can simply delete the node and fix the pointers of the node above the deleted one (these are O(1) const time operations). If the deleted node has 2 children, then we must replace it with the in-order successor. To find the in-order successor, I am using a helper function to find the deleted nodes right subtree left most node and performing a swap operation. This helper function makes the time complexity of removing a node with 2 children a O(log n + k) time complexity operation, where "k" is the number of nodes it takes to find the in-order successor.

Search ID:

1.) The time complexity of this function is O(log n) in worst case and O(1) space complexity.
2.) Where "n" is the number of nodes (accounts) in the AVL Tree.
3.) Reason: This function takes in an ID as a string and searches the tree to find that ID. If the ID is found, then the name is printed. This function takes advantage of the idea that the tree is sorted and balanced, making the AVL Tree symmetrical. This function checks if the ID is less than or greater than the root->ID and recursively traverses the tree until it finds the ID. Each iteration in the recursive calls, we are removing half of the possible outcomes leading to a O(log n) time complexity.

Search NAME:

1.) The time complexity of this function is O(n) in worst case and O(1) space complexity.
2.) Where “n” is the number of nodes (accounts) in the AVL Tree.
3.) Reason: This function takes in the name of a student and prints their ID. If there are multiple students with the same exact name, then it will print all those student’s IDs pre-order. Since the name attribute is not sorted inside the AVL tree, the function must travers every node in the tree, check/ compare the name with every node and print all the IDs with the same names in the pre-order arrangement. Because this is pre-order DFS, the function will recursively travers each node in the tree with the (check root, left, right) function calls, until the entire AVL tree has been checked leading to an O(n) time complexity. Also, no memory is being used.

Print In-order:

1.) The time complexity of this function is O(n) in the worst case and O(1) space complexity.
2.) Where “n” is the number of nodes (accounts) in the AVL Tree.
3.) Reason: Here, the function is just printing the name attribute stored in every node. This function performs a DFS in-order traversal. Each iteration the function recursively follows the “Left, print Root, Right” function calls to successfully iterate the entire tree, visiting and printing each nodes->name until there are no more nodes. Therefore, this is an O(n) time operation as it has to travers and print each node in the tree.

Print Per-order:

1.) The time complexity of this function is O(n) in the worst case and O(1) space complexity.
2.) Where “n” is the number of nods (accounts) in the AVL Tree.
3.) Reason: Here, the function is just printing the name attribute stored in every node. This function performs a DFS Pre-order traversal. Each iteration the function recursively follows the “print Root, Left, right” function calls to successfully iterate the entire tree, visiting and printing each nodes->name until there are no more nodes. Therefore, this is an O(n) time operation as it has to travers and print each node in the tree.

Print post-order:

1.) The time complexity of this function is O(n) in the worst case and O(1) space complexity.
2.) Where “n” is the number of nodes (accounts) in the AVL Tree.
3.) Reason: Here, the function is just printing the name attribute stored in every node. This function performs a DFS Post-order traversal. Each iteration the function recursively follows the “Left, right, print root” function calls to successfully iterate the entire tree, visiting and printing each nodes->name until there are no more nodes. Therefore, this is an O(n) time operation as it has to travers and print each node in the tree.

Print Level Count:

1.) The time complexity of this function is O(n) in the worst case and O(n) space complexity.
2.) Where “n” is the number of noes (accounts) in the AVL Tree.
3.) For this function, I am using the Breath First Search level order traversal algorithm where each node in the tree will be visited leading to a O(n) time operation. This function used 2 vectors. One vector stores the parent nodes, and the other vector stores the child nodes of the parent nodes that are in the first vector. The “level Count” increases when vector that stores parent nodes is empty, leading swap the vectors contents with the children to the vector that stored parents. And now, the process happens again till the parent vector is empty to then increment the “level count”, until the swap operation finds that there are no more child nodes to put in the parent vector, leading to the return value of “level count”.

   To wrap everything up: In each iteration of the while loop, we process all the nodes at the current level once. Leading the BFS function to visit each node exactly once. So, the time complexity is O(n), where “n” is the number of nodes in the AVL Tree.

   For the space complexity: I’m using 2 vectors to keep track of the nodes at the current level and the nodes at the next level. At any point in time, these vectors will contain at most all the nodes of the current level and all the nodes of the next level. This will lead to the worst case, when the tree is perfectly balanced, the maximum number of nodes at a single level is approximately half of the total number of nodes in the tree. So, the space complexity of the function is O(n/2) -> O(n), where “n” is the number of nodes in the AVL Tree.

Remove In-order N:

1.) The time complexity of this function is O(n) in the worst case and O(1) space complexity.
2.) Where “n” is the number of node (accounts) in the AVL Tree.
3.) In this function, we take in a integer value that acts as the index of the node we need to remove using the in-order DFS traversal algorithm. In the worst case, the index value that is passed in will be the last node in the tree for the in-order traversal. Making the function visit each node in the AVL tree with the “left, root, right” recursive function call until we have reached the in-order N node. When the node is reached, the function calls the remove ID function so that the node can be removed. The time complexity of this procedure is going to be O(1) constant time, as all we must do after this is just remove that node we found. Also, since no memory is being used, the space complexity is O(1).