Eric Fontes
3687-2820
Data Structures and Algorithms
Spring 2024

Gator AVL Tree Project Documentation

Time Complexity Analysis

> insert NAME ID
Time complexity: O(log(n))
Explanation:
The time complexity of the insert NAME ID command is O(log(n)) where n is the number of nodes in the tree. The command calls the public member function insert() which performs a few O(1) operations and calls the private member function insertAt(), so this function will determine the overall time complexity.

In this recursive function, there are 3 steps: BST insertion, height updating, and balancing. The BST insertion involves recursive calls that traverse the tree to find the correct leaf position for insertion. Due to the fact that the AVL tree is balanced, this process is equivalent to traversing a linked list with length H, where H is the height of the tree. Therefore the BST insertion is of O(H) time complexity.

After the node is inserted, the height of the parent node all the way up to the root node must be updated, once again similar to traversing a linked list of length H. Since updating the height is an O(1) operation and is repeated H times, the time complexity of height updating is O(H).

Finally, the node must be balanced. Calculating the balance factor is a simple subtraction of heights so it is O(1). The 4 rotation operations are all O(1), so balancing the tree is an O(1 + 1) ~ O(1) operation.

In total, the time complexity of insertAt() is O(H + H + 1) ~ O(2H). The height of the tree H is proportional to log(n) where n is the number of nodes. Therefore, the time complexity of insert is O(2log(n)) ~ O(log(n)).

> remove ID
Time complexity: O(log(n))
Explanation:
The time complexity of the remove ID command is O(log(n)) where n is the number of nodes in the tree. The command calls the public member function remove() which calls private member function removeNode(), so this function will determine the overall time complexity.

In the recursive function, there are 2 steps: traversal and removal. First, the node to be removed is found by recursively traversing through the tree. Since AVL trees are balanced, the worst case is where the node is located at a leaf position which is equivalent to traversing H nodes, where H is the height of the tree. Therefore, traversing is of O(H) time complexity.

Looking at deletion, zero and one child cases just involve copying a node, deleting a node, and returning a node, all of which are O(1) operations. For the two children case, there are a few O(1) operations and one O(H) find successor operation. Finding a node's successor involves continuously traversing through its sibling's left subtree. In the worst case, it will stop at a leaf node thus traversing the height of the tree, making this operation O(H).

In total, the time complexity of removeNode() is O(1 + H + H) ~ O(2H). The height of the tree H is proportional to log(n) where n is the number of nodes. Therefore, the time complexity of remove is O(2log(n)) ~ O(log(n)).

> search ID
Time complexity: O(log(n))
Explanation:
The time complexity of the search ID command is O(log(n)) where n is the number of nodes in the tree. The command calls the public member function searchID() which calls the private member function of the same name, so this private function will determine the overall time complexity.

The searchID() function works by recursively traversing through the tree until the node with the given id is found. Since AVL trees are balanced, the worst case is where the node is located at a leaf position which is equivalent to traversing H nodes, where H is the height of the tree. Therefore, searchID() is of O(H) time complexity.

The time complexity of searchID() is O(H). The height of the tree H is proportional to log(n) where n is the number of nodes. Therefore, the time complexity of search ID is O(log(n)).

> search NAME
Time complexity: O(n)
Explanation:
The time complexity of the search NAME command is O(n) where n is the number of nodes in the tree. The command calls the public member function searchName() which calls the private member function of the same name and stores the returned vector of nodes. After this, the elements of the vector are printed out. If k represents the number of nodes with a matching name, then the time complexity of search NAME will be O(k) + O(searchName()).

In the searchName() function, we traverse the entire tree to find matches for a name. This is because the name key isn't used for order in the tree, meaning that there is no way to tell if a certain name is in the left or right subtree of a node. Therefore, traversing the tree will take O(n) time. The insert() function is generally linear in the size of the vector, so it will take O(k) time.

Putting all of these together, the worst case time complexity of the search NAME function is O(k + k + n) ~ O(n + k). The worst case is when all nodes in the tree have the same name, so k = n. Therefore the final time complexity is O(n + n) ~ O(n).

> printInorder
Time complexity: O(n)
Explanation:
The time complexity of printInorder is O(n) where n is the number of nodes in the tree. This command involves traversing through all nodes in the tree where a single traversal step takes O(1) time. Therefore, the total time complexity of printing inorder will be O(n).

> printPreorder
Time complexity: O(n)
Explanation:
The time complexity of printInorder is O(n) where n is the number of nodes in the tree. This command involves traversing through all nodes in the tree where a single traversal step takes O(1) time. Therefore, the total time complexity of printing inorder will be O(n).

> printPostorder
Time complexity: O(n)
Explanation:
The time complexity of printInorder is O(n) where n is the number of nodes in the tree. This command involves traversing through all nodes in the tree where a single traversal step takes O(1) time. Therefore, the total time complexity of printing inorder will be O(n).

> printLevelCount
Time complexity: O(1)
Explanation:
The time complexity of printLevelCount is O(1) because it only involves calling getHeight(root) and displaying the result. Cout takes O(1) time and getHeight() just returns the height attribute of a node, an O(1) operation. Therefore, the total time complexity of printLevelCount is O(1 + 1) ~ O(1). It is independent of the number of nodes.

> removeInorder N
Time complexity: O(n)
Explanation:
The time complexity of removeInorder N is O(n) where n is the number of nodes in the tree. This operation involves calling inorderTraverse() which traverses through the tree and finds the Nth inorder node. Note that capital N represents the inorder index of the node to be removed. The id of the node is then passed into remove() which takes O(log(n)) time as discussed previously.

inorderTraverse() traverses through the tree with N steps and is therefore O(N) since each step is O(1). The total time complexity is O(N + log(n)). The worst case occurs when N = n - 1 (traverse inorder through the whole tree), so the worst case time complexity is O(n - 1 + log(n)) ~ O(n).

Reflection

> What did you learn from this assignment and what would you do differently if you had to start over?
This project strengthened my views on simplicity and modularization when coding. On the minesweeper project for COP3503, I often found myself confused when looking back at parts of my code. That's why I put emphasis on writing simpler and cleaner code when starting this project. I found that modularizing my code made it easier to debug and write my Catch2 test cases.

If I had to start over in this project, I would immediately create a separate class or namespace for handling input. Initially, I had everything in my main.cpp but that proved to be an issue when trying to parse commands in my Catch2 unit tests. I had to migrate my functions over to a new class and detangle the functionality from my main. Having insight into the Catch2 framework prior to coding my AVL tree class would have made it easier to write unit tests.