Arion Stern: AVL Tree Documentation

Time Complexity Analysis:

insert NAME ID

1. Variable Under Consideration

The variables under consideration are:

- **n**: The number of nodes in the AVL tree before insertion.
- **m**: The length of the name string

2. Definition of the Variables

- **n** represents the number of nodes in the AVL tree. Since the tree is self-balancing, the height of the tree remains O(log n). This height determines how deep an insertion can go before reaching a leaf node.
- **m** represents the number of characters in the name string. The regex validation function must scan the entire string to verify that it contains only letters and spaces, which affects complexity.

3. Justification of Complexities

1. **Validation Complexity:**
   - The ufid validation is a fixed-length regex check (std::regex_match(ufid, std::regex("[0-9]{8}"))), which takes O(1) time.
   - The name validation scans all m characters (std::regex_match(name, std::regex("^[A-Za-z\\s]+$"))), making it O(m).
2. **Duplicate Check Complexity:**
   - The function checkDuplicate() calls searchByIDHelper(), which performs a binary search on the AVL tree.
   - Since the AVL tree remains balanced, the worst-case search complexity is O(log n).
3. **Insertion Complexity:**
   - insertHelper() searches for the correct position in the tree, which takes O(log n) in a balanced AVL tree.
   - Creating a new node and inserting it is O(1).

4. **Balancing Complexity:**
    - The updateHeight() function updates the height of ancestor nodes. Since we traverse back up to the root, this takes O(log n).
    - The getBalance() function is called on each visited node, making its total time O(log n).
    - If an imbalance is detected, at most one rotation (single or double) is performed, each taking O(1).

Final Worst-Case Complexity: O(m) + O(log n)

remove ID

1. Variable Under Consideration

The variable under consideration is:

- n: The number of nodes in the AVL tree before deletion.

2. Definition of the Variable

- n represents the total number of nodes in the AVL tree before the removal operation begins. Since an AVL tree remains balanced, its height is O(log n). The depth of the node being removed impacts the time complexity, as the function must traverse the tree to locate it.

3. Justification of Complexities

1. **Duplicate Check Complexity (O(log n))**
    - Before deletion, checkDuplicate() is called to verify whether the given ID exists in the tree.
    - checkDuplicate() calls searchByIDHelper(), which performs a binary search in the AVL tree.
    - Since the AVL tree is balanced, the search takes O(log n).
2. **Search and Removal Complexity (O(log n))**
    - The removeHelper() function is a recursive binary search that navigates to the node that matches the given ID.

    - Since the height of an AVL tree is O(log n), searching for the node takes O(log n) in the worst case.
    - Deletion has three scenarios:
        - No children: Delete the node in O(1).
        - One child: Replace the node with its child in O(1).
        - Two children: Find the inorder successor using findMin(), which requires an O(log n) traversal to the leftmost node in the right subtree.
3. **Balancing Complexity (O(log n))**
    - After deletion, the updateHeight() function updates the heights of ancestor nodes, which takes O(1) per node but propagates up to the root, leading to O(log n).
    - The getBalance() function is called for each ancestor node, adding another O(log n) factor.
    - If the tree becomes unbalanced, at most one rotation (single or double) is performed, each taking O(1).

Final Worst-Case Complexity: O(log n)

- Since all operations (searching, deleting, updating heights, and balancing) operate within the height of the tree, the worst-case complexity remains O(log n).

search ID

1. Variable Under Consideration

- n: The number of nodes in the AVL tree before searching.

2. Definition of the Variable

- n represents the total number of nodes in the AVL tree before the search begins. Since an AVL tree is balanced, its height remains O(log n), which determines the maximum depth that must be traversed to locate a node.

3. Justification of Complexities

1. **Checking if the Input is a Valid ID (O(1))**
    - The function first verifies whether the input is an 8-digit numeric ID using std::regex_match(input, std::regex("[0-9]{8}")).
    - Since the regex pattern is fixed (only 8 characters), this runs in O(1).
2. **Searching for the Node (O(log n))**
    - The function calls searchByIDHelper(), which is a binary search in the AVL tree.

    - Since the tree is balanced, the worst-case scenario occurs when we traverse the entire height of the tree, which is O(log n).
    - If the node is found, its associated name is printed in O(1). If not found, "unsuccessful" is printed in O(1).

Final Worst-Case Complexity: O(log n)

- The dominant term in this function is the binary search operation, making the worst-case complexity O(log n).

search NAME

1. Variable Under Consideration

- n: The number of nodes in the AVL tree before searching.
- m: The length of the name string being searched.

2. Definition of the Variables

- n represents the total number of nodes in the AVL tree before searching begins. Since an AVL tree is balanced, its height remains O(log n), but the worst case may require visiting every node.
- m represents the length of the name string, which is processed to remove quotes.

3. Justification of Complexities

1. **Checking if the Input is a Name (O(1))**
    - The function first checks whether the input is enclosed in quotes using input.front() == '"' && input.back() == '"'.
    - This is an O(1) comparison.
2. **Extracting the Name (O(m))**
    - The function extracts the name using input.substr(1, input.size() - 2).
    - This operation takes O(m) because it creates a new substring of length m - 2.
3. **Searching for Matching Names (O(n))**
    - searchByNameHelper() is called, which performs a preorder traversal of the entire AVL tree to find matching names.
    - Since names are not ordered like IDs in a BST, we must check every node in the worst case, leading to O(n) complexity.
4. **Printing the Results (O(k))**
    - If matches are found, each UFID is stored in a vector and printed.

    - If there are k matches, printing takes O(k), but since k ≤ n, this does not change the overall worst-case complexity.

Final Worst-Case Complexity: O(m) + O(n)

printInorder

1. Variable Under Consideration

- n: The number of nodes in the AVL tree before traversal.

2. Definition of the Variable

- n represents the total number of nodes in the AVL tree. Since an in-order traversal must visit every node exactly once, the number of nodes determines the time complexity.

3. Justification of Complexities

1. **In-order Traversal (O(n))**
    - printInorder() calls getInorderNames(), which recursively performs an in-order traversal
    - Every node in the tree is visited once, making this operation O(n).
2. **Storing Results in a Vector (O(n))**
    - Each node's name is added to a std::vector<string> inside getInorderNames().
    - Since every node is processed once, this step takes O(n).
3. **Printing Results (O(n))**
    - printNames() iterates through the vector and prints each name, taking O(n) time.

Final Worst-Case Complexity: O(n)

printPreorder

1. Variable Under Consideration

- n: The number of nodes in the AVL tree before traversal.

2. Definition of the Variable

- n represents the total number of nodes in the AVL tree. Since a pre-order traversal must visit every node exactly once, the number of nodes determines the time complexity.

3. Justification of Complexities

1. **Pre-order Traversal (O(n))**
    - printPreorder() calls getPreorderNames(), which recursively performs a pre-order traversal
    - Every node in the tree is visited once, making this operation O(n).
2. **Storing Results in a Vector (O(n))**
    - Each node's name is stored in a std::vector<string> inside getPreorderNames().
    - Since every node is processed once, this step takes O(n).
3. **Printing Results (O(n))**
    - printNames() iterates through the vector and prints each name, taking O(n) time.

Final Worst-Case Complexity: O(n)

printPostorder

1. Variable Under Consideration

- n: The number of nodes in the AVL tree before traversal.

2. Definition of the Variable

- n represents the total number of nodes in the AVL tree. Since a post-order traversal must visit every node exactly once, the number of nodes determines the time complexity.

3. Justification of Complexities

1. **Post-order Traversal (O(n))**
    - printPostorder() calls getPostorderNames(), which recursively performs a post-order traversal
    - Every node in the tree is visited once, making this operation O(n).
2. **Storing Results in a Vector (O(n))**
    - Each node's name is stored in a std::vector<string> inside getPostorderNames().
    - Since every node is processed once, this step takes O(n).
3. **Printing Results (O(n))**
    - printNames() iterates through the vector and prints each name, taking O(n) time.

Final Worst-Case Complexity: O(n)

printLevelCount

1. Variable Under Consideration

- n: The number of nodes in the AVL tree before execution.

2. Definition of the Variable

- n represents the total number of nodes in the AVL tree. Since an AVL tree is balanced, its height is O(log n), which determines the result of this function.

3. Justification of Complexities

1. **Checking if the Tree is Empty (O(1))**
    - The function first checks whether this->root is nullptr.
    - This is a simple O(1) operation.
2. **Getting the Height of the Tree (O(1))**
    - The function calls getHeight(this->root), which simply returns the stored height value of the root node.
    - Since AVL trees store height as an attribute and update it during insertions and deletions, retrieving it is an O(1) operation.
3. **Printing the Height (O(1))**
    - The height value is printed directly using std::cout, which runs in O(1).

Final Worst-Case Complexity: O(1)

- Since printLevelCount() only retrieves a stored value without traversing the tree, its time complexity is O(1).

removeInorder N

1. Variable Under Consideration

- n: The number of nodes in the AVL tree before execution.
- N: The position of the node to be removed in inorder traversal.

2. Definition of the Variables

- n represents the total number of nodes in the AVL tree. The function must perform an inorder traversal to locate the Nth node.
- N is the index of the node to be removed in the inorder sequence, ranging from 0 to n-1.

3. Justification of Complexities

1. **Edge Case Check (O(1))**
   - The function first checks whether N is out of bounds using if (N < 0 || N >= countNodes(this->root)).
   - Since countNodes() is called, we need to analyze its complexity.
2. **Counting the Nodes (O(n))**
   - The function calls countNodes(this->root), which recursively counts all nodes in the tree.
   - This requires a full traversal of the tree, making it O(n) in the worst case.
3. **Finding the Nth Node in Inorder Traversal (O(n))**
   - The function calls findNthInorder(this->root, N, ufid), which performs an inorder traversal while keeping track of the index.
   - Since inorder traversal visits all nodes in the worst case, this operation takes O(n).
4. **Removing the Found Node (O(log n))**
   - After retrieving the UFID of the Nth node, the function calls remove(ufid), which internally calls removeHelper().
   - removeHelper() searches for the node, taking O(log n) in a balanced AVL tree.
   - The removal process (including balancing) also runs in O(log n).

Final Worst-Case Complexity: O(n) + O(log n) → O(n)

- The dominant term is the O(n) inorder traversal, making the overall complexity O(n).

Reflection:

What did you learn from this assignment, and what would you do differently if you had to start over?

I learned a lot about the implementation and design of AVL trees and their self-balancing properties. I also gained experience in time complexity analysis, recursive tree traversals (inorder, preorder, and postorder traversals), memory management(ensuring efficient allocation and deallocation of nodes), command line and input stream functions, and regex validations. If I were to start over, I would have done a few things differently; for instance, I would have made my main functions have return values other than void (I was under the assumption they were supposed to be void but learned when I was practically done with the assignment that this was not the case). This would have made it much easier to write my unit tests and would have reduced the need for some helper functions and improved code modularity. Also, since removeInorder() requires a full inorder traversal (O(n)), I would explore alternative approaches, such as maintaining an auxiliary structure to speed up the process and reduce traversal time to O(log n). I would have implemented more test cases early on in my project as it would have saved a lot of time debugging later in the process which would not have been necessary if I caught certain issues earlier. Along with this, I would have included some debugging statements in the early stages of the project just to validate proper input and outputs.