Efficiency Analysis - Program 5
Name: Evelyn Nguyen
Program: CS302 Program #5
Date: 12/5/2025

Effiency Write Up

For Program 5, I implemented a Red-Black Tree to store and manage social media accounts. So, in thhis writeup I explains why I chose this data structure, analyzes the time and space complexity of my operations, and discusses the trade offI made.

The biggest and hardest decision I had to make was choosing between a 2-3-4 and a Red-Black Tree. I decided to use a Red-Black Tree because it self-balances, which guarantees O(log n) time for insertions, retrievals, and other operations.. The Red-black Tree prevents this by using rotations and color properties to keep the tree balanced. The height is always guaranteed to be at most 2 * log(n+1), which means operations stay fast even with many accounts.

I also considered using an AVL tree like I originally planned in my specification. AVL trees are more strictly balanced than Red-Black trees, which means retrievals would be slightly faster. However, Red-Black trees require fewer rotations during insertion. And also, since my program does more insertions than retrievals, especially when users are adding multiple accounts at the start, so I think the Red-Black Tree was the better choice.

In term of the time complexity of each operation in my program. The insert operation is O(log n) because it first does a recursive BST insertion by traveling down the tree height, which takes O(log n) time. Then it calls the fix_insert method to restore Red-Black properties, which also takes O(log n) in the worst case because it might travel up to the root. The rotations themselves are O(1) each, and we do at most 2 rotations per insertion. So the total is O(log n) + O(log n) = O(log n).

The retrieve operation is also O(log n). It recursively searches down the tree, comparing usernames at each node. In the worst case, it travels the full height of the tree, which is O(log n). I made usernames case-insensitive by converting to lowercase during comparison, which adds O(k) time where k is the username length, but since usernames are typically short (5-15 characters), this is negligible compared to the tree traversal.

The display_all operation is O(n) because it must visit every single node in the tree using in-order traversal. There's no way to make this faster because we literally need to display every account. Each node is visited exactly once, and creating the display string for each account takes constant time, so the total is Θ(n), which means it's exactly n operations, not just bounded by n.

The count_by_type and get_total_followers operations are also O(n) because they traverse the entire tree. I know these could be optimized to O(1) by maintaining counters that update during insertion. For example, I could have three variables like instagram_count, facebook_count, and tiktok_count that increment whenever I insert

an account of that type. This would make count_by_type instant, but it would add complexity to the insert method and use extra memory. Since statistics are displayed less frequently than insertions in my program, I decided the O(n) traversal was acceptable.

For space complexity, each node in my tree stores a SocialMediaAccount object plus the node overhead (color, three pointers for left, right, and parent). This means each node uses O(1) space. The total space for the tree is O(n) where n is the number of accounts. Each account also stores lists for posts, reels, groups, etc., so the actual space depends on how much data each account has. An Instagram account with 100 posts uses more memory than one with 2 posts. But overall, the tree structure itself is O(n).

And at first, I used Python lists for storing posts, reels, and groups instead of NumPy arrays. Lists are dynamic and grow automatically, which is perfect for variable-length data. NumPy arrays would require preallocation or expensive resizing. I only convert to NumPy when calculating statistics with get_engagement_stats, which takes place once per display. And I believe this trade-off prioritizes flexibility over memory efficiency.

One bottleneck in my program is that every time I want to view an account, I have to call retrieve(), which is O(log n). If users view the same accounts repeatedly, I could add caching using a dictionary to store recently accessed accounts. The first lookup would still be O(log n), but subsequent lookups would be O(1). However, this adds complexity and uses extra memory.

Overall, I learn a lot in this program. The Red-Black Tree guarantees O(log n) for all single-account operations, which stays fast even as data grows. The recursive implementation is clean and doesn't cause performance problems because the tree stays balanced.

VSCODE WRITE-UP

For this assignment, I used the VS Code debugger consistently while developing and testing my program. To me,one of the most helpful features was the ability to set breakpoints exactly where I suspected the logic might fail. This let me pause execution and check the values of my variables in real time, instead of printing everything manually as I usually did in the past

I also used the “Run and Debug” panel to watch how input data moved through different methods, especially when I was tracking down issues with invalid types or negative values. When I built the test suite for prog5, I sometimes ran individual tests with the debugger attached so I could see why a specific test was failing. Overall, using the VS Code debugger made the whole assignment less overwhelming for the

most part. It did saved me a lot of time, helped me verify my assumptions, and gave me more confidence that my code was behaving correctly