Worst case time complexity analysis of each command:

Before we can start analyzing commands we need to first explain the difference between expected worst case and absolute worst case time complexity because the foundation I build my AVL Tree on, is a hash map, and hash maps are probabilistic in nature. Absolute worst case time complexity is the computation time of the worst possible occurrence. For the hash map I used for my AVL tree, this worst possible occurrence is so rare, a person is more likely to win the lottery on their first ticket than for the worst case to happen, unlike the standard C++ library's unordered map. This makes Funnel Hashing a safer hashing method against hackers as hackers will struggle to find program inputs that will degrade the hash map to this worst case. In all other cases, the time it takes to execute a function is the expected worst case time complexity.

We are going to analyze the underlying structure of this hybrid hash map AVL tree data structure first, which is located in the FunnelHashMap header file, to make the AVL tree method analysis more concise and understandable:

The insert method:

- has an absolute worst case time complexity of O(N), where N is the input size. My project assumes default and constant values for the many different variables that have complex relationships with others (alpha, beta, delta, load_factor), so we can drop them from the final complexity. Since the load factor is constant, the time complexity, table size * load_size * all the other greek alphabet variables that depend only on load size = N becomes table size = N. Since N is the only variable left and nearly all elements can collide on an insertion in the worst case, the final complexity is O(N).
- has an expected worst case time complexity of O(1). The paper "Optimal Bounds for Open Addressing Without Reordering" available at: https://arxiv.org/abs/2501.02305, provides that time complexity as $O(\log^2(1/\delta))$, where δ is 1-load_factor. This simplifies as in my implementation, load factor is constant: $O(\log^2(1/(1\text{-constant}))) = O(\log^2(1/(\text{constant}))) = O(\log^2(\text{constant})) = O(\text{constant}) = O(1)$.

The find method for both existent and non-existent key search shares the same complexity as the insert method, but their reasons are different. For non-existent key searches, the process is identical to insert. For existent key searches, the process is nearly identical to insert as empty and deleted keys are skipped over. This doesn't affect the worst case because the distribution of keys in the hash map varies on the input data, so there could be a case of no empty or deleted keys on the probe sequence. If a key is found or not, it returns an iterator, which is an O(1) operation.

The erase method is identical to the find method, but instead of returning an iterator of the found value, it defaults the value and updates the attributes, which are O(1) operations.

Note: for all FunnelHashMap header file operations, if the constant multiple of the table size was larger than N, which is not a concern for this project because the maximum number of nodes is known ahead of time and we can set that to N, the table would need to be rehashed and all time complexities for both expected and absolute worst case would degrade to O(N) for common operations of the funnel hash table. This is because rehashing could require moving all the elements inserted (input size) inside the table to another clean copy of the table.

Now we will analyze the command methods. The commands that call functions in the AVL tree and validate inputs are O(1), not including the command formatting and for loop to call each line. All of these methods are located in the AVL Tree header file:

The insert method:

- has an absolute worst case time complexity of O(N*log(N)) and has an expected worst case time complexity of O(log(N)), where N is the input size, because the insert method:
    - we ignore everything outside the while loop because the loop executes a variable number of times, including find and insert as they are also in the loop.
    - requires a while loop when the node isn't the root, which requires calls of the find and insert method in my chosen hash table (analysis provided in common methods): O(N) for absolute worst case and O(1) for expected worst case. Other assignments, branches, and methods are O(1), except for the balance tree method as it requires O(1) amount of calls to find, but nothing more. Same situation with the helper functions of the balance tree method.
        - The number of loop iterations is equal to the length of the longest path in the tree, which will always be O(log(N)) in an AVL tree. This is because in the worst case when you have an almost completely full AVL tree, the height of the tree is O(log(N)), and the height is the number of nodes you have to travel to get to the bottom of the tree. Traveling up the tree to rebalance is also required, but it's the same distance as traveling down, and O(log(N)) + O(log(N)) = O(2*log(N)) = O(log(N)).
- The while loop iterates O(log(N)) times and the most expensive method: the find method is O(N) for the absolute worst case and O(1) for the expected worst case, leading to the final time complexities stated above.

The remove method:

- has an absolute worst case time complexity of O(N*log(N)) and has an expected worst case time complexity of O(log(N)), where N is the input size, because the remove method:
    - has one while loop in the two children case and one while loop at the end for balancing the tree. No other cases have loops, but they do call the find method in my chosen hash table (analysis provided in common methods): O(N) for absolute worst case and O(1) for expected worst case. :
        - The two children case loop is to find the inorder successor which could involve traveling down the longest path of the tree, which is O(log(N)). As it

travels down the longest path of the tree, it has to call the find function each time, so the absolute worst case is O(Nlog(N)) and the expected worst case is O(log(N)).

- The balance tree function loop is the same as insertion: the absolute worst case time complexity is O(Nlog(N)) and the expected worst case time complexity is O(log(N)).
- The erase function is not as computationally intensive as the loops. It has the same complexity as the find function (analysis provided in common methods).

- The largest time complexities of the remove function are O(Nlog(N)) for the absolute worst case and O(log(N)) for the expected worst case, so that is the time complexity.

The printPreorder, printInorder, and printPostorder functions:

- have an absolute worst case time complexity of O(N^2) and expected worst case time complexity of O(N), where N is the number of nodes in the tree. This is because those functions must visit every node in the tree, and additionally non-existent nodes, but it never visits more non-existent nodes than N. The find function (analysis provided in common methods): O(N) for absolute worst case and O(1) for expected worst case is called N times. This leads to the final absolute worst case time complexity because O(N) N times = O(N^2), and final expected worst case time complexity of O(N).

The searchByKey and searchByKeyPrint functions:

- has an absolute worst case time complexity of O(N), where N is the input size, and expected worst case time complexity of O(1). This is because the search by key function is at most two find functions: (analysis provided in common methods): O(N) for absolute worst case and O(1) for expected worst case. We remove the constant that is two.

The searchByValue and searchByValuePrint functions are identical to the PrintInorder function, but return or print values instead of keys. This means the time complexity is O(N^2) for the absolute worst case and O(N) for the expected worst case.

The printLevelCount function:

- has an absolute worst case time complexity of O(N) and expected worst case time complexity of O(N), where N is the number of slots in the hash map. This is because all slots in the hash map including deleted and unoccupied have to be visited to find the max height value of all nodes.

The removeInorder function:

- has an absolute worst case time complexity of O(N^2) and expected worst case time complexity of O(N), where N is the number of nodes in the AVL tree. This is because a full traversal of the entire tree must be performed in the worst case, which is identical to printInorder: an absolute worst case time complexity of O(N^2) and expected worst case time complexity of O(N), where N is the number of nodes in the tree. However, it differs in that when a certain node index is reached, the remove function is called, which has an absolute worst case time complexity of O(N*log(N)) and has an expected worst case time complexity of O(log(N)), where N is the input size, but these time complexities are from O(N^2) and O(N), which are respectively lower than the full traversal, so they are dropped from the final time complexity.

As you might notice, the time complexities for this hybrid data structure are much worse for the absolute worst case, but that could be a worthy tradeoff for skipping downward traversal for practically always O(1) access to any node and faster deletion, halving memory requirements as it combines two data structures into one, and better cache utilization as a this hash map is less scattered than operating system memory.