Time complexities

- insert NAME ID
    - O(log n) where n is the number of nodes in the tree
    - The recursive call travels down one path of the tree and therefore touches at most log(n) elements and the balancing algorithm traverses back up this same path of at most log(n) elements
    - O(2 log n) simplifies to O(log n)
- remove ID
    - O(log n) where n is the number of nodes in the tree
    - The recursive call travels down one path of the tree and therefore touches at most log(n) elements and the balancing algorithm travels back up this same path of at most log(n) elements
    - O(2 log n) simplifies to O(log n)
- search ID
    - O(log n) where n is the number of nodes in the tree
    - The recursive call travels down one path of the tree and therefore touches at most log(n) elements
- search NAME
    - O(n) where n is the number of nodes in the tree
    - The recursive call touches each node exactly once and the print queue touches each node exactly once in the worst case where all nodes in the tree have the search name (O(2n) simplifies to O(n))
- printInorder
    - O(n) where n is the number of nodes in the tree
    - The recursive call touches each node exactly once and the print queue touches each node exactly once
    - O(2n) simplifies to O(n)
- printPreorder
    - O(n) where n is the number of nodes in the tree
    - The recursive call touches each node exactly once and the print queue touches each node exactly once
    - O(2n) simplifies to O(n)
- printPostorder
    - O(n) where n is the number of nodes in the tree
    - The recursive call touches each node exactly once and the print queue touches each node exactly once
    - O(2n) simplifies to O(n)
- printLevelCount
    - O(1) constant time regardless of number of nodes n in the tree

    - The function does not involve any looping or searching through elements of the tree, so it is constant time
- removeInorder N
    - O(log n + N) where n is the number of nodes in the tree and N is the inorder index of the node being removed
    - The recursive traversal call touches exactly N nodes in order to reach the Nth node and the conducts the “remove ID” (see above) algorithm, which has a time complexity of O(log n) where n is the number of nodes in the tree

Reflection

- What I learned
    - From this project, I have developed a far better understanding of how AVL trees and other balanced trees work behind the scenes. Visualizing rotations and sketching out how they change a tree, which once seemed strange and difficult to understand, now feel hardly different something like manually writing out long division
    - Very often I found myself creating helper functions that involved recursively conducting a traversal or binary search through the tree structure. I used to mix up the types of traversals and had a very difficult time understanding how to interpret and create recursive algorithms, but this project has certainly developed my understanding of these concepts. I can now confidently say that I can conduct traversals and design recursive algorithms as it applies to my future projects in the computer science field
- What I would do differently
    - Perhaps the biggest thing that I would do differently is start sooner. I drastically underestimated the amount of time that would need to be put forth towards debugging unusual behavior and creating test inputs and outputs for edge cases.
    - I would also make sure to put more effort upfront into increasing the modularity of my code.
        - Within main.cpp, I could have benefitted greatly from creating helper functions that would handle data validation. This would have made my code cleaner and easier to debug
        - Within my AVL tree file, I could have more efficiently written my helper functions so that instances of using the same traversal could have been consolidated into a single helper function, rather than creating specialized traversal functions for each application that required them