Project 1 Documentation - Hunor Vajda

Time Complexities

- **insert - O(h + n)**
  - h - the height of the tree
  - n - number of nodes
  - There are at most h comparisons before method reaches valid place to insert new node
  - Assume constant time for comparisons
    - All IDs are 8 chars long
  - After insertion call, method checks balance factors for each node in call stack
    - Assume balance factor calculation and rotations are O(1)
  - Once insert helper function is done, calcHeight function is called which is O(n)
    - Calls on each node of tree once with constant time comparisons/ calculation

- **remove - O(h + n)**
  - h - the height of the tree
  - n - number of nodes in the tree
  - In the worst case, the method makes (h-1) comparisons to find a node with two children
    - With two children, method needs to find inorder successor with an inorder traversal
      - In worst case the traversal is O(n)
      - Inorder successor can't have 2 children

- **search ID - O(h)**
  - h - height of the tree
  - Searches until first and only matching ID
    - At most h comparisons
    - Assuming constant time comparison

- **search NAME - O(n)**
  - n - number of nodes in tree
  - Must go through all nodes in a pre-order traversal
  - Constant time to check for nullptr
  - Once search helper call is finished, method must print all found IDs
    - Worst case, all nodes have matching name so creating string to return is O(n)
      - Assuming constant time to modify string each time

        - Not nested so only adding time complexities O(2n) simplifying to O(n)

- **printInorder - O(n)**
    - n - number of nodes in tree
    - Must access all nodes in tree
    - Constant time to check nullptr
    - Once print helper call is finished, method must print all names in vector (all the names in tree)
        - Assuming constant time to modify string each time
        - Not nested so only adding time complexities O(2n) simplifying to O(n)

- **printPreorder - O(n)**
    - n - number of nodes in tree
    - Must access all nodes in tree
    - Constant time to check nullptr
    - Once print helper call is finished, method must print all names in vector (all the names in tree)
        - Assuming constant time to modify string each time
        - Not nested so only adding time complexities O(2n) simplifying to O(n)

- **printPostorder - O(n)**
    - n - number of nodes in tree
    - Must access all nodes in tree
    - Constant time to check nullptr
    - Once print helper call is finished, method must print all names in vector (all the names in tree)
        - Assuming constant time to modify string each time
        - Not nested so only adding time complexities O(2n) simplifying to O(n)

- **printLevelCount - O(n)**
    - n - number of nodes in tree
    - Calls calcHeights for each node in the tree
        - Assume each call is constant time (n times)
    - After calcHeights recursive call stack is resolved, root node height is accessed and returned which is O(1)

- **removeInorder - O(N + h + n)**
    - N - input of removeInorder function
    - h - height of tree
    - n - number of nodes in tree
    - Must go through N of the inorder traversal steps to find node to delete
        - Once the node is found, call the remove ID function for which the complexity was already discussed

What did I learn and what would I do if I had to start over?

I learned to be more consistent with my coding style and that I should test as I write. If I had to start over, I would start with the command parsing so I could fully test each function as I was writing them with commands instead of writing the AVL tree methods one by one in main.