GatorAVL Project Documentation

A. Complexity Analysis

1. Insert: O(nlogn); n := number of nodes in tree

The insert function first calls search by ID and returns false if the ID is already found in the tree. The search by ID function requires O(logn) time in the worst case, where n is the number of nodes in the tree. If the ID is not found, insert then calls insertHelp, a recursive function which calculates node heights and balance factors and conducts rotations as necessary after inserting the new node. In the worst case, the path from the newly inserted node to the root will be as long as the height and each node will require a rotation. Because the height of a balanced tree is proportional to logn, insertHelp will conduct rotations on a maximum of logn nodes. For each of these nodes, insertHelp uses a utility function called height to calculate balance factors and identify the conditions for rotations. This function takes O(n) time to find the height of each subtree and return the maximum, making insert's overall time complexity O(nlogn).

T(insert) = logn + n*logn = 2nlogn ~ O(nlogn) time complexity, worst case.

2. Remove by ID: O(logn); n := number of nodes in tree

The remove by ID function is a BST-only implementation that does not rebalance the tree after the removal, but per the instructions we assume the tree is balanced before and after the removal. Like insert, remove first calls search, which takes O(logn) time in the worst case, and returns false if the ID is not found. Remove then calls removeHelp. If the node to be deleted has two non-null children, it must be replaced by its in order successor, or the minimum element in the right subtree. In the worst case, this replacement will have to be made at each level of the tree, thus requiring O(logn) time.

T(removeID) = logn + logn = 2logn ~ O(logn) time complexity, worst case.

3. Remove by n: O(n); n := number of nodes in tree

This function removes the nth node from an in order traversal of the tree. The function first conducts an in order traversal and saves each node in a vector in that order. This requires O(n) time, where n is the number of nodes, since every node in the tree must be accessed. The nth node is found and its ID is passed to remove. The vector access takes constant time, and remove by ID takes O(logn) time as discussed above, where n is again the number of nodes in the tree.

T(removeN) = n + logn ~ O(n) time complexity, worst case.

4. Search by ID: O(logn); n := number of nodes in tree

Because the nodes in the tree are sorted by ID, the search by ID function only has to recursively search one of two subtrees at each call. This means that search by ID only makes one comparison at each level of the tree. Since a balanced tree's height is proportional to logn, where n is the number of nodes in the tree, search by ID will conduct O(logn) comparisons in the worst case.

5. **Search by name: O(nm); n := number of nodes in tree, m := length of target string**

   Because the nodes in the tree are not sorted by name, each node in the tree must be checked to complete a search by name. At each node, the search by name function conducts a string comparison, returning matching names to a vector, and calls itself recursively on each subtree. String comparison requires O(m) time, where m is the length of the target string passed in as the parameter. The string comparison must be done n times for each of the n nodes, so the search by name function requires O(nm) time. However, if the number of nodes in the tree is very large compared to the length of the average target string, the m factor in O(nm) will begin to act like a constant. I expect this will be case in this scenario – the number of students in the university greatly exceeds the average length of a name, and student name length probably centers around a mean value with relatively low variance.

   T(searchNAME) = n * m = O(nm)

6. **Print In Order**
7. **Print Pre Order**
8. **Print Post Order**

   6.-8. O(n); n := number of nodes in tree

   Each of the three print functions relies on a traversal, either in order, pre order, or post order, to access and print the associated name for each node in the tree. Since each node must be accessed, all three print functions require O(n) time to execute, where n is the number of nodes in the tree.

9. **Print Level Count: O(n); n := number of nodes in tree**

   The print level count function returns the value of the height utility function called on the root of the tree. Height recursively checks the height of each node's subtrees and returns the maximum. Because it must access each node, height requires O(n) time, where n is the number of nodes.

B. Reflection

Through this project, I became more familiar with AVL tree properties, implementing rotations and traversals, and writing recursive functions. I also gained experience analyzing the impact of my design choices on my functions' time complexity. If I were to do the project again, I would rewrite insertHelp to rely on a height value stored in each node instead of calling the height function each time to calculate balance factors. While I made an initial effort to implement the stored height process, I switched to using the height function when it didn't quite work instead of working to debug it. Using the stored height process would improve the time complexity of insertHelp from O(nlogn) to O(logn). I would also incorporate the searching step in insert and remove into the recursive help functions. The recursive functions essentially conduct the search again in order to find the correct place to insert or the correct node to remove, and the bases cases can account for instances when the ID is already in the tree or is not found. I used a separate searching step for simplicity, so insert and remove could clearly return false when appropriate. Lastly, I would implement an AVL version of remove that rebalanced the tree as necessary. Although this step was optional, I know it would significantly expand my understanding of the AVL tree data structure and recursive functions.