Project 5 Writeup

Part 1: Performance Analysis

Note: Below is a performance analysis of every method (public or Private) that changed in time complexity from the b-tree of project 4.

__Balanced(self,t)- Because there are only 5 possibilities for a given node(balanced, left-left, left-right, right-right, and right-left) and the children's heights are known, the operation to detect the type of rotation is constant. Once the type of rotation is discerned, the operation to rotate is also constant because the nodes to be rotated are predefined along with their heights. Thus, regardless of the size of the tree, this method is constant in time or O(1).

__check_imbalance(root)- This method checks the child nodes to see if the tree is balanced. If it is not balanced, it returns the type of imbalance, left or right. As the heights of the child nodes are known, this method has a time complexity of O(1) or constant.

__sub_check_imbalance(root)-This method checks the larger child of a root node of an imbalanced tree to see if it needs to rotate before the root node is rotated. Similar to __check_imbalance(root), the heights of the child nodes are known so this method runs in O(1) time.

__rotate_right(root)- This method rotates a tree if It needs to be rebalanced. It rotates the nodes right and then adjusts their heights. Because the rotation operation follows a strict formula where the number of nodes that need to be adjusted stays the same, this method runs in constant time. It is also important to mention that this method uses the __get_height(root) method which is also constant. Therefore, the big O complexity for this method is O(1).

__root_left(root)- This method does the same exact thing as __rotate_right except all the nodes are switched. Consequently, this method runs in O(1) time.

Insert_Element(self,root)- Unlike the b-tree in project 4, this method runs in O(log(n)) time. The reason for this is because the balance method forces all trees to have a child height balance of no greater than 1. This means that the tree must be near full before it can grow in height. Thus, instead of growing in worst-case at O(n) nodes, the tree grows at O(log(n)) nodes. This means that, in order to find the location of insertion of a given node, the deepest possible level of recursion would be O(log(n)). Therefore, insertion gives us logarithmic time.

remove_element(self,root)- In the same way that insertion requires O(log(n)) time, removal does the same for an AVL tree. The number of times the recursive removal function would be activated based of the height of the tree which is O(log(n).

to_list(self)- Similar to the inorder, preorder, and postorder methods, this method has a worst-case time complexity of O(n^2). The reason for this quadratic time is because the number of nodes that needs to be appended is equivalent to n, and the append function in python takes O(n) time. For each n in a tree, we must perform another linear time method to append it to the list. Thus, the time complexity for this method is quadratic.

Part 2: AVL Tree Sorting

The AVL tree has significant benefits to a traditional Binary search tree. The improvements in performance come from the fact that the tree grows in height in O(log(n)) time vs O(n) time which is significantly more efficient for search, insertion, and removal. The reason that this sorting approach is superior to the traditional binary search tree is because the balancing of the tree does not take extra time as it runs in constant time. With this, we gain the benefit of O(log(n)) search which improves the time complexity of the insertion and removal methods. Without any loss in time complexity from balancing the tree, we gain a lot of time complexity in insertion and removal. The only downsides of this kind of tree is that it does not optimally sort every kind of data set. If we want some very specific functionality in from our tree, an AVL tree is not always the best choice. Not every decision or set of data is based on a binary set of rules. What if we wanted to have three child nodes or a heap of nodes? In the case of the AVL tree, the best-case height is the upper bound of log(n+1)-1, and the worst-case height is the lower bound of log((n+1)/2). Overall, this isn't bad, and it is quite a clever approach to optimizing search time complexity.