Gator AVL – Documentation

Time Complexity

Insert *NAME ID*

Best Case	Ω(1)	Where constant is a single letter name and insert at the root
Average Case	Θ(m + log(n))	Where m is the length of the string name iterated and n is the number of nodes recursively travelled
Worst Case	O(m + n)	Where m is the length of the string name iterated and n is the all the number of nodes that are recursively traversed if unbalanced

Remove *ID*

Best Case	Ω(1)	Where constant is removing the Student at the first root
Average Case	Θ(log(n))	Where n is the number of nodes traversed recursively as the root moves left or right along the tree
Worst Case	O(n)	Where n is the number of nodes traversed recursively if the root moves through all of the tree (i.e. unbalanced)

Search *ID*

Best Case	Ω(1)	Where constant is finding the Student at the first root
Average Case	Θ(log(n))	Where n is the number of nodes traversed recursively
Worst Case	O(n)	Where n is all the nodes traversed if unbalanced

Search *NAME*

Best Case	Ω(1)	Where constant is only one/no nodes in the tree to be searched and string.compare() has complexity of 1 since name is a single letter
Average Case	Θ(n + m)	Where n is all the nodes of the tree recursively passed in pre-order and m is the complexity of string.compare() as it iterates through each letter in the root name
Worst Case	O(n + m)	Where n is all the nodes of the tree recursively passed in pre-order and m is the complexity of string.compare() as it iterates through each letter in the root name

PrintInorder

Best Case	Ω(1)	Where constant is only one/no nodes in tree traversal
Average Case	Θ(n)	Where n is all the nodes in the tree recursively traversed
Worst Case	O(n)	Where n is all the nodes in the tree recursively traversed

PrintPreorder

Best Case	Ω(1)	Where constant is only one/no nodes in tree traversal
Average Case	Θ(n)	Where n is all the nodes in the tree recursively traversed
Worst Case	O(n)	Where n is all the nodes in the tree recursively traversed

PrintPostorder

Best Case	Ω(1)	Where constant is only one/no nodes in tree traversal
Average Case	Θ(n)	Where n is all the nodes in the tree recursively traversed
Worst Case	O(n)	Where n is all the nodes in the tree recursively traversed

PrintLevelCount

Best Case	Ω(1)	Where constant is finding the height of an empty / singular tree
Average Case	Θ(n)	Where n is all the nodes traversed to calculate the height for each root side and find the max
Worst Case	O(n)	Where n is all the nodes traversed to calculate the height for each root side and find the max

RemoveInorder *N*

Best Case	Ω(1)	Where constant is the time to push only one node / return empty node into a stack and perform removal
Average Case	Θ(n)	Where n is all the nodes in the tree pushed to the stack and added into a vector, to then be removed by index
Worst Case	O(n)	Where n is all the nodes in the tree pushed to the stack and added into a vector, to then be removed by index

Reflection

Based on this project, I was able to get a better understanding of how to handle recursive function calls and the flow of a pointer-based data structure. By debugging each call on the stack frame, I was able to gain insight into how each of the functions are chained together and the logical recursion of aa AVL tree creation. It helped me better understand where the specific data structure member functions should be used and parameters that it might take.

If I were to do the assignment differently, it would be to give myself ample more time to experiment with different applications of the functions. Rather than be primarily recursion-based, how would an iterative solution look like? I would also like to improve the time complexity for a lot of the main and helper functions. Most ended up being O(n) due to naively iterating through each element. And so, I would look into seeing if there is a more efficient library or construction that could be used. Otherwise, generally clean up the code and maintain readability.