Mauricio Benjamin Mossi

Project 1 Gator AVL Project.

Time Complexity Analysis.

Insert [name][id]: Let ***n*** denote the number of nodes in the tree. For the insert command the insertion time depends on the height of the tree, which is proportional to at most $\log_2 n$ for AVL trees, and the student ID, which we store as a string and is bounded by its length of 8 digits. When we perform an insertion on an AVL tree we use the student ID as a key. Since all student ID's have the same length, namely 8-digit characters, I will take the comparison of student ID strings for insertion as constant, as at most the comparison compares 8 characters.

For an insertion we must compare our node with the root node of the current subtree. If our key is greater or smaller than the root and the root has children on the right if greater and on the left is smaller, we call the function recursively with the rightmost node of the root or leftmost node of the root respectively. If the root does not have children, we insert the node on the right if it is greater or on the left is smaller. We do so recursively for the subtrees necessary until insertion or until we find a duplicate, in which case our operation fails. After each call to the insert function, we go down a level on the tree, and we know that at most there exist $\log_2 n$ levels. Per level we perform at least 2 comparisons which we take as constant, and or an insertion which also is constant (just assigning pointers).

After the insertion we check if the invariants of the AVL tree are satisfied, namely if the balance factors of -1, 0, +1 are maintained. To do so we calculate the height difference of the node as we pop each ancestor node insert call of the stack. As mentioned previously, at most there will be $\log_2 n$ ancestors. The calculation of the height difference is also constant since we only get the height of the children's nodes and perform the necessary arithmetic. If it is the case that we are violating the invariants of the AVL tree we will need to perform rotations. Rotations are just the reassignment of pointers, so we take them as constant. At most we will perform $\log_2 n$ rotations since the only height that changes are that of the ancestors and there are at most $\log_2 n$ ancestors.

Therefore, the time complexity of insertion in the tree $O(\log_2 n)$.

Remove [id]: Let ***n*** denote the number of nodes in the tree. Similarly to insertion, the removal time of a node depends on the height of the tree, which is proportional to at most $\log_2 n$ AVL trees, and the student ID, which we store as a string and is bounded by its length of 8 digits. When we perform a removal on an AVL tree we use the student ID as a key. Since all student ID's have the same length, namely 8-digit characters, I will take the comparison of student ID strings for removal as constant, as at most the comparison compares 8 characters.

For a removal, we also compare the right and left children of the root of a subtree to find the node. The process is like the one outlined above for insertion; the only difference is that if we find the node, we perform the removal process. There are four cases for removal. If we do not find the node, then we go down all the levels of the tree, which as stated above are $\log_2 n$ at most. If we find the node, there are three cases. If the node has no children, the case is trivial, we just set the parents pointer to the node to null and deallocate the node, both constant operations. If we find the node and it has one child subtree

the case is again trivial, we perform the same operation but now the parent of the node points to the subtree, we can do so because the subtree must be strictly greater than or lesser than the parent of the node to be deleted. If we find the node and it has two children's subtrees, then we "swap" the node with its successor and delete the successor position in the tree. To do we must perform a traversal to find the successor, the traversal searches for the leftmost descendant of the right child of the node to be deleted, this traversal does not perform any comparisons as it just searches for the left most node on the subtree which is at most $C - \log_2 n$, where C is the number of levels we have already traversed to find the node to be deleted, in the worst case $C = 0$. Once the successor is found we return a pointer to the successor, change both the key and the value of the node with that of the successor, and we call remove with the key of the successor on the right child of the node, this operation is guaranteed to be either case a) no children, or b) one children both being $\log_2 n$ in the worst case, therefore this removal in general takes at most $2\log_2 n$ or $O(\log_2 n)$.

For our specific implementation we chose to perform rotations on deletion also, which as outlined above are proportional to the height of the tree which is $\log_2 n$.

Therefore, the time complexity of remove in the tree $O(\log_2 n)$.

Search [id]: Let ***n*** denote the number of nodes in the tree. For the search [id] command the search time depends on the height of the tree, which is proportional to at most $\log_2 n$ for AVL trees. This is specific to this case where we search based on the key of the nodes which are based on the id.

For a search we must compare our node with the root node of the current subtree. If our key is greater or smaller than the root and the root has children on the right if greater and on the left is smaller, we call the function recursively with the rightmost node of the root or leftmost node of the root respectively, if our node's key is equal to the root's key or equal to the child or the root then we print the name of the node with that id. It is important to note that unlike id, name has no restriction on its length, therefore the printing time linear and not constant across nodes, so we will take ***m*** to be the length of the longest name in the tree. If the node is not found, we print the message "unsuccessful", which we will assume to be a constant operation.

Since at most we will travel $\log_2 n$ nodes and print at most ***m*** characters, in the worst case our operation will take $O(\log_n 2 + m)$.

Search [name]: Let ***n*** denote the number of nodes in the tree. For the search [name] command the search time depends on the number of nodes in the tree and the length of the name provided to search, which we will denote as ***m***. Since names are not keys on our tree, we will have to traverse the whole structure to find all occurrences of the name provided. To do so we use a preorder traversal of all nodes in the tree. Per each node in the tree, we will compare if its value matches the name provided. The maximum number of character comparisons performed per node is ***m*** + 1, since it could be the case that both names match up to the m^{th} character, but the value of the node has more than ***m*** characters, in that case there is not a match. In practice, ***m + 1*** comparisons might render out of bounds errors and would be inefficient. Instead, we compare lengths, which is a constant operation, and if they do match, then we do character-based comparison which would render a total of ***m + 1*** operations.

If the name is found, we print the id, which as mentioned above is an 8-digit character string, which we will take as constant, and we do so ***k*** times, which is the number of nodes with the same name. However, even though at most ***k = n***, it would still be considered a constant operation performed while traversing the tree. If the node is not found, we print the message "unsuccessful", which we will assume to be a constant operation.

Since we always iterate through all ***n*** nodes, and per each we perform at most ***m + 1*** comparisons, in the worst case our operation will take $O(n + (m + 1))$.

PrintInorder: Let ***n*** denote the number of nodes in the tree. For the printInorder command the time depends on the number of nodes in the tree and the length of the names in the tree, since as mentioned above, there is no restriction on the size of the names. For printInorder we traverse all nodes in order and print their respective names. Inorder traversal of an AVL tree is linear with the number of nodes, and the printing of names can be approximated as linear with the number of characters of a name. Let's denote ***m*** as the length of the longest name.

Therefore, the time complexity of printInorder is $O(n * m)$

PrintPreorder: Let ***n*** denote the number of nodes in the tree. For the printPreorder command the time depends on the number of nodes in the tree and the length of the names in the tree, since as mentioned above, there is no restriction on the size of the names. For printPreorder we traverse all nodes in order and print their respective names. Preorder traversal of an AVL tree is linear with the number of nodes, and the printing of names can be approximated as linear with the number of characters of a name. Let's denote ***m*** as the length of the longest name.

Therefore, the time complexity of printPreorder is $O(n * m)$

PrintPostorder: Let ***n*** denote the number of nodes in the tree. For the printPostorder command the time depends on the number of nodes in the tree and the length of the names in the tree, since as mentioned above, there is no restriction on the size of the names. For printPostorder we traverse all nodes in order and print their respective names. Postorder traversal of an AVL tree is linear with the number of nodes, and the printing of names can be approximated as linear with the number of characters of a name. Let's denote ***m*** as the length of the longest name.

Therefore, the time complexity of printPostorder is $O(n * m)$

PrintLevelCount: For printLevelCount we just print the height of the root node if it exists, else we print zero. This is because the node-based height of the tree equals the level count of the tree.

Therefore, the time complexity for printLevelCount is $O(1)$

RemoveInOrder [n]: Let ***n*** denote the number of nodes in the tree and ***m*** as the index n provided to the removeInorder command, this is done to follow the convention above of using ***n*** as the number of nodes in the tree. For the removeInOrder command the time complexity only depends on the index ***m*** and the underlying removal time complexity in an AVL tree, which we determined to be $O(\log_2 n)$. For our implantation we augmented our AVL tree and added a size private property whose value can be accessed via the size method. The size property allows us to know the bounds, in terms of size, of the AVL tree. If the index ***m*** is greater than the size – 1 we know the index is out of bounds and can print unsuccessful. If the index is within bounds, we perform an inorder traversal up to the m^{th} node and return a pointer to the node. With the pointer to the node, we call remove on the node's key. It is important to keep in mind that even though we are removing the m^{th} node it does not entail that we only iterate m^{th} elements. For example, to reach the 0^{th} element in an inorder traversal we need to traverse down the height of the AVL tree, which we know is $O(\log_2 n)$. On the other hand, to reach the n^{th} element we indeed must go through n nodes in an inorder traversal. Therefore, we will take the time complexity of the initial traversal as linear with the number of nodes.

Therefore, the time complexity for removeInorder is $O(n + \log_2 n)$.

Remarks about assignment.

The assignment was a formative experience:

- I learned and came to appreciate the importance of invariants. If you know a condition is invalid within an object, you only need to check it once and the rest of the code can perform operations without unexpected results.
- I came to appreciate the importance of error when: the object invariants could not be established and to avoid silent errors. It is better to terminate the program than to continue with undetermined behavior.
- I learned the importance of flexible and extensible APIs. I built an AVL template class which other classes specialize for their specific use case uses as its underlying data structure. The AVL class takes care of the invariants of an AVL tree and other classes just consume the data structure.
- I learned and relearned some features of C++. As standouts, I really enjoyed working with lambdas and ***std::functional*** to make generic preorder, postorder, and inorder methods for the AVL template class, similar to forEach methods in languages like JavaScript and Kotlin.
- I came to appreciate recursion and understand it more in depth. For example, I learned and experienced how, within a recursive function, code placed after the recursive call is evaluated as the call stack unwinds, allowing you to "bubble" up results.
- I learned and experienced how data structure augmentation can greatly improve a data structure. For example, I added a size property to the AVL template class, which allowed me to keep track of its size. Therefore simplifying operations as bounds checking.
- I came to appreciate the properties of functions, namely local returns. I used local returns within the inputHandler functions to significant code duplication.

If I had to start over, I would allocate more time to learn supporting technologies like Makefiles and CMake which greatly simplify development by, for example, enabling the construction of build and test scripts.