1 Documentation

What is the computational complexity of each method in your implementation? Reflect for each scenario: Best, Worst and Average.

I will regularly reuse the same variable to mean the same item, for example l will always mean the length of the ID or NAME passed in, whichever is larger (string length); similarly, n will be the number of nodes in the tree.

1. `insert NAME ID` Best case for inserting a StudentNode into the tree is $O(l)$ (where l is the length of the ID or NAME, whichever is larger, and assume they grow at the same rate for the whole project). This constitutes checking through each character of the NAME and ID and finding that the NAME/ID are invalid. Otherwise when they are both valid, and the ID already exists as a node in the tree, we may get $O(l+\log(n))$ complexity where n is the number of nodes in the tree (due to having to search for the duplicate).

   In the average case, the node does not exist in the tree and the node to be inserted will be valid (the NAME and ID are valid), in which case the node is inserted. After insertion we call the `balance` method which is quadratic complexity $O(n^2)$ where n is the number of nodes in the tree, because we traverse through every node in the tree, and compute the balance factors of each node from the bottom up (leaves to root) in the recursion at every call. Every time we compute the balance factor we have to traverse through all of the nodes in the subtree at the location the balance factor number is needed (for large n we stil get around $O(n)$ complexity). So in balancing every call takes $O(n)$ to determine the balance factor, and then since we do this for every node, the balancing method is around $O(n^2)$ in complexity. Hence overall the insertion method in the average case is $O(l + n^2)$

   Because balancing requires traversing through the whole tree anyways in my implementation, the worst case is still the same, $O(l + n^2)$ complexity where n is the number of nodes in the tree.

2. `remove ID` The best case scenario is again going to be $O(l)$ when the ID is invalid or $O(l + \log(n))$ where n is the number of nodes in the tree but we failed to find the ID in the tree if it was valid.

   The average case involves a valid ID and finding the node, and so after successfully removing the node in $O(l + \log(n))$ time, we again rebalance the tree (even though this was completely optional). Hence this adds quadratic complexity as mentioned before, so then again we get $O(l + n^2)$ complexity for removal in the average case.

   The worst case is identical since we again have to call the balancing method, so the complexity is still $O(l + n^2)$.

3. `search ID` In the best case we get $O(l)$ complexity when the ID is invalid, and then if the ID is valid it could be possible to get $O(l)$ complexity for then having to search (the tree is a BST!) for the node and find that it is in the root already.

   In the average case we would have to search through the majority of the tree, which would cause the search method to have a complexity of $O(l + \log(n))$.

The worst case is identical, since the tree is balanced.

4. `search NAME` In the best case the NAME is invalid, so we take $O(l)$ complexity to terminate.

   In the average case the NAME is valid, so then we retrieve the inorder traversal of the nodes in a vector and traverse through the whole list to find a few (meaning around constant) matches. Both of these take $O(n)$ complexity, so overall the complexity ends up being $O(l + n)$.

   In the worst case every name in the tree matches, so in my implementation all such matches (the whole tree's traversal) gets copied into another vector and we print off all of the IDs. Conveniently, copying whole arrays is the same complexity as traversing through the array, and so we get the same complexity as in the average case, just arguably taking more real life time. We get $O(l + n)$ complexity for checking validity of NAME and then n for traversing through the inorder traversal twice.

5. `printInorder` There is no distinguishing feature of the tree that would lend itself to a more/less efficient traversal of all of the nodes. For this reason, all of the cases are identical. In the best case, average case, and worst case you *must* traverse through all of the nodes in the tree, and in my implementation I actually just put all of the nodes in a vector and traverse through the vector to print all of the values out. To get the nodes in the vector I used a recursive solution which was also $O(n)$ since we recursively traveled to each node *once* in the tree (true for the other traversals as well). Hence we have complexity $O(n)$, where n is the number of nodes in the tree.

6. `printPreorder` Identical to `printInorder`.

7. `printPostorder` Identical to `printInorder`.

8. `printLevelCount` This method is only sightly different from the previous three, but we achieve the same result. In the implementation for this method we traverse through each level and put all of the children of the nodes in each level into a vector which will at the end contain all of the nodes in the tree. This takes $O(n)$ complexity, since every pass through the loop we would have to basically see each child twice. Then at the end we print out all of the NAMES in the nodes in the final vector, containing all n nodes, which is also $O(n)$ complexity. This does not change based on any change to the data, we still have to traverse through all of the nodes. So in all cases the complexity is $O(n)$ where n is the number of nodes in the tree.

9. `removeInorder N` We would first get the inorder traversal of the nodes in the tree, which as before we said takes $O(n)$ complexity, but then we would seek to remove the N-th item in the list if it exists.

   So in the best case N is not a number which is valid, so not in the indicial bounds of the vector of nodes. Hence we do not do any removals and so the complexity stays at $O(n)$.

   In the average case N is a valid number and so we can remove the N-th item or student from the vector and thus from the tree, and we already determined the complexity of removing a student from a tree to be $O(l + n^2)$ since we rebalance the tree at the end (this supersedes the complexity of getting the inorder traversal). Hence the complexity in the best case is $O(l + n^2)$ where n is the number of nodes in the tree and l is the length of the NAME of the N-th node (we may discount l if we like, since we know it is finite length).

The worst case scenario is identical to the average case, there is not really a difference in the tree which would worsen the complexity (traversals and removals are invariant).

2 Reflection

What did you learn from this assignment and what would you do differently if you had to start over?

The most important thing I learned from this assignment was that the simplest solution was the best solution. I came up with various ways to handle the more complicated methods, but ultimately went with solutions which resembled the pseudocode shown in the lectures, which were very clean compared to my first or second attempts.

I also learned that I should have probably written an outline before proceeding with the project, because design wise I had to make a few changes along the way which were minor but I could have avoided dealing with design changes during implementation.

Aside from lessons, some concrete things I learned from working on this assignment were how to navigate documentation, at least better than before. I read more about how regular expressions worked despite not actually using the `regex` header in my project, but I also read about related ways to determine if a `char` was alphanumeric or not.

Dealing with recursion was one of the more difficult parts of the project, partially because I started out with more complicated implementations for what should have been simpler, and I felt that I got a lot of good practice in for dealing with solutions that use recursion. If I had to start over, I might approach recursive solutions more concretely with a whiteboard or scratch paper so I could trace the recursive calls/returns.

One last (slightly embarrassing) thing that I learned for the first time was how to use breakpoints, since before I had never really thought to use them since bugs in the past were very minor, but when so many methods involved recursion I finally decided it was time to debug better than before. I should have spent more time learning how to debug more effectively in the past before starting this project.

Citations for the project:

Kapoor, A. (2021). Trees-1. Gainesville, FL; University of Florida.

Kapoor, A. (2021). Trees-3. Gainesville, FL; University of Florida.

Kapoor, A. (2021). Trees-4. Gainesville, FL; University of Florida.