Writeup for Project 4

Part 1: Performance Analysis

This program constructs a new data structure: trees, which consists of a root, leaves, and inner nodes. A tree is a connection of private nodes with attributes left and right. In this implementation, one node can have at most two children -- one left child and one right child. The following is the performance of each method in the tree class.

The first method in my tree implementation is the update height method. It is a private method that updates the height of a node after other methods are called. It compares the height of that node’s children and updates its height to be the height of its higher children plus one. If one of its children point to None, its height would just be the height its other children plus one. Any leaf node would have height 1. This method has constant time performance as it just compares two values and makes arithmetic calculations.

The next method is the private recursive insertion method used to insert elements into the tree. It compares the new value to a specified existing element and determines whether it should be inserted into that element’s left subtree or the right subtree. After that, it would call itself again to insert the element into that subtree and make another comparison. If it finds an element of the same value, a value error exception would be raised as duplicate values are not allowed in the structure. When the method finds an empty space, it would create a new node and return that node so that it can be connected to the structure. Each time the method is called, it updates the height of each node with the update height function discussed above. Ultimately it would return the current node so that it is reconnected to the structure. In this implementation, as the tree is unbalanced, the method has a linear worse case performance explained below.

```
1
 \
  2
   \
    3
```

Consider the following tree. Insert 1, 2, 3 into the tree in this order. 1 would be the root and 3 would be the only leaf node. Suppose I want to insert a new element 4 into the tree, it would be the right child of 3 as 4 is larger than all the elements in the tree. The recursive function would be called 4 times until it finds an empty space to insert the new element. Each time it would also use the constant time performance update height function to update the height of each node. So in total, it has a linear O(n) performance. If the tree is perfect or complete, on the other hand, it would have an O(log(n)) performance as it would only have to go through log(n) nodes in order to reach the leaf position.

The public insert function is the function that users have access to. It calls the private recursive insertion function and updates the root of the tree to be the node returned by the recursive function. This method has linear time performance as it calls the linear time recursive function discussed above and updates the root.

The next function is the small value locator. It finds the smallest value in a tree structure starting from a certain node. As small values are located in the left part of the tree, it locates the first node without a left child recursively. This function has linear time worst-case performance as it may have to go through every element to find the smallest. Consider the following example.

```
    3
   /
  2
 /
1
```

Insert 3, 2, 1 into the tree. 3 would be the root and 1 would be the only leaf. If the function is called, it would have to go through 3 elements to find the smallest element 1 -- the first element without a left child. Thus the performance would be linear O(n).

One of the most important and complex methods is the private recursive removal method that removes an element with a given value. Each time it is called, it compares the value with the current element and decides whether the value should be in the left or right subtree of that element. Then it calls itself again to compare the value with the root of that subtree. When it finally locates the value, it would consider 3 situations. If the element containing the value has no children, return nothing so that the element is disconnected from the structure. If the element has 1 child, return that child so that the element is unlinked. If the element has 2 children, call the small value locator function to locate the smallest value in its right subtree, replace the value to be removed by the smallest value. Then call the recursive removal method again to remove that duplicate smallest value in the subtree. Each time the recursive removal function is called, it also calls the update height function to update the height of each node. This method has a linear time performance which will be illustrated by the following two examples.

```
1          1           1               2               2
 \          \         / \             / \             / \
  2    →     2       0   4           0   4           0   4
   \                    /      →        /      →        /
    3                  3               3               3
                      /               /
                     2               2
```

In the left example, insert 1, 2, 3 into the tree. 1 is the root and 3 is the only leaf. If I want to remove 3, the element would have to go through all the elements from 1 to locate 3 and then remove it from the structure. So the performance is linear. In the right example, insert 1, 4, 3, 2, 0 into the tree. If I want to remove 1, the function would first locate 1. Then the small value locator function would be called, it would go through 4, 3, 2 to locate the smallest value 2. Then the recursive function would call itself to delete 2 starting from 4. So this case is actually even more complex than the last example, resulting in even longer processing time. But the overall performance is linear as the method would go through part of the element twice so the performance is still O(n).

The next part is the private recursive in-order traversal method of the tree. It returns an empty string if the node it is operating on is none. If the node is not None, it calls itself again to get the string representation of the in-order traversal of the node's left child, add the node's own value to the string representation with string concatenation, then add the inorder traversal of the node's right child to the string. It also adds any specific format requirements. Finally, it returns the overall string. This method has a linear time performance. The method would perform on each element in the string to get its in-order traversal. And each time it is called, it would add 3 strings together. So the overall performance is O(n). Consider the following example.

```
    3
   /
  2
 /
1
```

In order to get the in-order string representation of this tree, the function would be called for 3 times. It would first get the in-order representation of 1 by adding two empty strings and '1' together. Then it would add '2' and an empty string to '1'. Finally, it would add '3' and an empty string to the final string and return the string representation. Here the representation is [1, 2, 3]. So the method is used on every element and every time it is called, 3 strings are added together, resulting in O(n) performance.

The public in-order function is used by the users to get the in-order string representation of the tree. It calls the recursive in-order function and adds specific formats to the string returned. This method has a linear performance as it uses the in-order traversal function discussed above.

The private recursive pre-order and recursive post-order function are similar to the recursive in-order string representation. They just concatenate strings in different orders, thus both have the same linear performance.

The public pre-order and post-order function are also similar to the public in-order function, they call the recursive pre-order and recursive post-order functions respectively, thus resulting in the same linear time performance.

The get height function returns the height of the root. If the root points to None, it returns 0. This method has constant time performance as the height of the root is stored as an attribute thus the function just looks up and returns the value.

Part 2: Test Cases Explanation

The test cases mainly focus on three user interference methods, insertion, removal, and get height. These are the 3 crucial methods ensuring the correct functionality of the Tree class.

First test the string representation of an empty tree in various order. In-order, pre-order and post-order traversals should all return an empty bracket ‘[ ]’. Then test the insertion method. Use string representation in 3 different orders to ensure that the structure of the tree is correct. For, example, insert 1, 4, 2, -1, the structure of the tree would be:

```
     1
    / \
  -1   4
      /
     2
```

And the corresponding string representation would be: in-order [-1, 1, 2, 4], pre-order [1, -1, 4, 2], post-order [-1, 2, 4, 1]. Test insertion into an empty tree, a tree of heigh one, height two, height 3 respectively. For each height, first insert a duplicate value and see whether the program raises a ValueError and leave the tree unchanged. Then, test insert into incomplete trees, complete trees and perfect trees of that height respectively.

For each type of tree, also insert elements into different positions to ensure there are no errors when inserting into various tree structures. After each insertion, test the structure of the tree with 3 different traversals. Also, utilize the get height function to ensure that the insertion function correctly implements the height of the tree structure. Note that these tests also suffice to show that different traversals are working correctly to get the right string representation as any mistake in traversal methods can be detected because we know the structure of the tree. As no 2 different trees have the same 3 traversals, these tests suffice to show that the insertion method correctly inserts elements to the right position in trees of all heights and structures. This test provides a solid foundation for test of removals.

Then, the removal method is tested. First, test remove from an empty tree. A Value Error should be raised as no element is in the tree. Then test removing from a tree of heigh one, height two, height 3, height 4 respectively. For each height, first remove a non-existent value and see whether the program raises a ValueError and leave the tree unchanged. Then, test remove nodes at different positions of that height respectively. Remove nodes at root, leaf and inner position to ensure that the removal function works correctly to remove nodes at various positions in the structure. Also, test remove nodes with 1 child, 2 children, and 0 children respectively, ensuring different portions of the recursive removal function all works correctly. Each time after removing an element, utilize the get height function to ensure that the removal function correctly implements the height of the tree structure. As no 2 different trees have the same 3 traversals, these tests suffice to show that the remove method correctly removes elements from the right position in trees of all heights and structures. It also shows that after each removal, the structure of the remaining tree is the same as desired.

As each time the get height function is used to test the height of the tree after insertion or removal, its functionality can also be assured as it correctly returns the height of various tree structures. So the get height function also functions correctly, returning the right height to the user.

Note that in some parts of the test, in order to form a correct tree structure for testing, both insertion and removal functions are used arbitrarily. For example, in order to form a tree with structure:

```
   1
  / \
 0   2
```

```
    3                                      3
  /  \                                    /  \
-1    4                 3               -1    6                1
       \        →     /   \        →     / \           →      /  \
        5           -1     6           0   1                 0    2
         \                                   \
         6                                   2
```

First insert 3, 4, 5, -1, 6 into the structure, then remove 4, 5 from the tree, add 1, 2, 0 to the tree, remove 3, 6, -1 so the structure is finally as above. This can ensure that the insertion and removal function correctly interact with each other and updates the height of each node in the tree.

In all, all the test cases work together to ensure that each method correctly interacts with the overall tree structure.