Writeup for Project 5

Part 1: Performance Analysis

This program constructs a new data structure: AVL trees, which consists of a root, leaves, and inner nodes. An AVL tree is a connection of private nodes with attributes value, left and right, same as a node in Binary Search Tree. In this implementation, one node can have at most two children -- one left child and one right child. The difference between a Binary Search Tree and an AVL Tree is that an AVL Tree is always balanced, which results in logarithmic look up time. AVL tree has two attributes -- root and element_number, element_number records the number of elements in the tree which is useful in constructing the list representation of the tree. The following is the performance of each method in the Balanced Binary Search Tree class.

The only difference between an AVL Tree and a Binary Search Tree is that it contains a balance function that can rotate the tree so that the whole structure is balanced. This involves several private methods. The first one is the balance calculator function that calculates the balance factor of a given node. The method calculates the balance factor by subtracting the height of the left child from the height of the right child. If any of that node's children is None, that child has height 0. Then the function returns the balance. This function has a constant time performance as height is stored as an attribute in the node class. All the function does is retrieve the value and apply arithmetic which results in constant time performance.

The second 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 of 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.

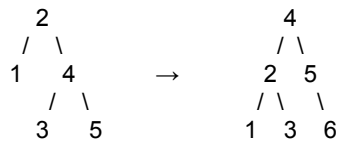
The third method is rotate left which rotates the subtree rooted at a given node to the left. It first records the right child of that old root of the subtree to be the new root. Then it makes the new root's original left child be the right child of the old root. After that, It makes the old root the left child of the new root, and the rotation is done. The final step is to update the height of the new root and the old root and return the new_root. This method has constant time performance as all it does is reconnect 3 nodes. The update

height function called in the method also has constant time performance as explained in the last paragraph. So the rotate left method has an overall O(1) performance.
The next method is rotate right which rotates the subtree rooted at a given node to the right. This method is similar to the rotate left method. It first records the left child of that old root of the subtree to be the new root. Then it makes the new root's original right child(floater) be the left child of the old root. After that, It makes the old root the right child of the new root, and the rotation is done. It also updates the height of the new root and the old root and returns the new_root. This method also has constant time performance as it reconnects 3 nodes just like the rotate left function. The update height function called in the method also has constant time performance as explained in the last paragraph. So the rotate right method has an overall O(1) performance.

The balance function is the main function that coordinates all the above methods. It initially calls the balance calculator function to calculate the balance of a given node. If the balance factor is 2 and the subtree at that node is right heavy, it calls the balance calculator again to check the balance of that node's right child. If the balance is smaller than 0 which means a double rotation is needed, it first rotates right the subtree rooted at the right child, then it rotates left the whole subtree rooted at the given node. If the balance is greater or equal to 0, it rotates the whole subtree to the left once. On the other hand, if the balance factor is -2 and the subtree is left heavy, it then checks the balance factor of the node's left child. If the balance is greater than 0 which means double rotation is needed, it first rotates left the subtree rooted at the left child, and then it rotates right the whole subtree rooted at the given node. If the balance of the child is smaller or equal to 0, it rotates the whole subtree to the right once. If the balance is smaller than 2 and greater than -2, meaning the structure is balanced, it updates the height of the given node and returns the node. The balance function has a constant time performance. It checks the balance of a node and conducts the correct rotation according to the. As both rotation functions, the balance calculator function and update height function all has a constant time performance, the balance method also has a constant time performance as it uses these methods at most two times when it is called. So the overall performance is constant time O(1).

All the above methods are designed to make the tree balanced, the following methods are the regular tree methods that are similar to the ones in Binary Search Tree. The First 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 right subtree. After that, it calls 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

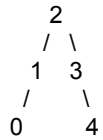
duplicate values are not allowed in the structure. When the method finds an empty space, it would create a new node and return that node so that it can be connected to the structure. Each time the method is called, it updates the height of each node with the update height function discussed above. Then it would call the balance function to make the subtree rooted at the current node a balanced tree. Finally, the node is reconnected to the whole structure. In this implementation, as the tree is balanced, the method has a logarithmic worse case performance as explained below.



Consider the following tree. Insert 1, 2, 3, 4, 5 into the tree in this order. 2 would be the root, 1, 3, 5 would be leaf nodes, and 5 would be the only inner node. Suppose I want to insert a new value 6 into the tree, it would be the right child of 5 as 6 is larger than all the existing elements. 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 balance function to make sure the tree is balanced. So in total, it has an $O(\log_2(n))$ performance as each time we decide a path almost half of the elements in the tree will not be considered. This is true because of the balanced nature of the tree.

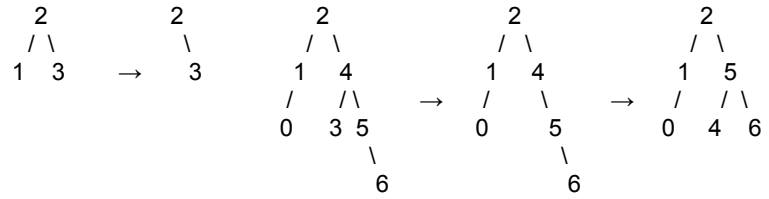
The public insert function is the function that users have access to. It calls the private recursive insertion function and updates the root of the tree to be the node returned by the recursive function. This method has $O((\log_2(n))$ performance as it calls the logarithmic time recursive function discussed 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 $O(\log_2(n))$ worst-case performance as the tree is balanced. Consider the following example.



Insert 1, 2, 3, 4, 0 into the tree. 2 would be the root and 0,4 would be leaves. If the function is called, it would have to go through 3 elements to find the smallest element 0, the first element without a left child. As each time the function chooses a path to go, almost half of the element in the subtree will not be considered, thus the worst-case performance would be $O(\log_2(n))$.

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 balance function to make sure each subtree is balanced. This method has an $O(\log_2(n))$ performance which will be illustrated by the following two examples.



In the left example, insert 1, 2, 3 into the tree. 2 is the root and 1, 3 are leaves. If I want to remove 1, the function would have to go through 2 to remove 1 from the structure. So the performance is $O(\log_2(n))$ because if I choose to go to the left child of 2, all the elements in the right subtree will not be considered. And as the tree is balanced, the elements in the left and right subtree are almost the same. In the right example, insert 1, 2, 3, 4, 5, 0, 6 into the tree. If I want to remove 2, the function would first locate 2. Then the small value locator function would be called, it would go through 4, 3, to locate the smallest value 3. Then the recursive function would call itself to delete 3 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 still $O(\log_2(n))$ as the function may have to go through $O(\log_2(n))$ elements to locate the smallest value and another $O(\log_2(n))$ elements to delete that value. So the performance of recursive removal is $O(\log_2(n))$.

The public removal function is the function that users have access to. It calls the private recursive removal function and updates the root of the tree to be the node returned by the recursive function. This method has $O((\log_2(n))$ performance as it calls the logarithmic time recursive removal function discussed above and updates the root.

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.

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' to '1'. Finally, it would add '3' and two empty strings 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 public to list function returns the list representation of the in-order traversal of the AVL Tree. It first initializes a list containing Nones whose length is equivalent to the number of elements in the tree. Then it sets a variable current which stands for list index to be 0. Finally, it calls the recursive to list function and returns the generated list.

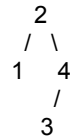
The private recursive to list function inserts values in the AVL Tree to the list created by the public to list function recursively. It takes in a node as a parameter. If the given node is None, the function would do nothing and return. If the node is not None, the function would first call itself to insert the node's left child into the list. Then, it would put the node's own value to the correct index in the list. The index is initialized to be 0 and each time a value is inserted, the index is grown by 1. Ultimately the function calls itself again to add the node's right child into the list. Note that this recursive function has no return value as it only alters the list initialized by the public function.

The private recursive to list function has a linear time performance as it has to go through every element to retrieve the value. Then it replaces the element at a certain index in the list with that value which requires constant time. So the overall performance is linear time. The performance of the public to list function also has a linear time performance as it initializes a list and a variable then calls the recursive function.

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 four user interference methods, insertion, removal, to_list 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, post-order traversals and the to_list method 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, 2, 4, 3, the structure of the tree would be:



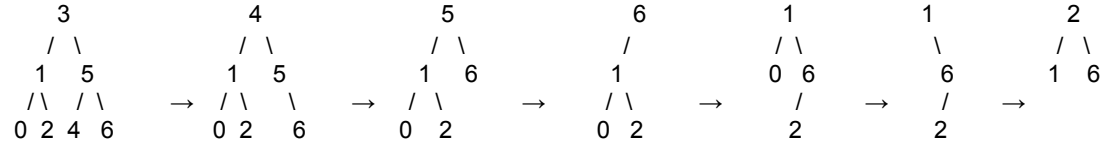
And the corresponding string representations would be: in-order [1, 2, 3, 4], pre-order [2, 1, 3, 4], post-order [1, 3, 4, 2]. Test insertion into an empty tree, a tree of heigh one, height two, height 3 respectively. For each height, first insert a duplicate value and see whether the program raises 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 so that various double and single rotations will take place. This ensures that the algorithm does not raise errors when inserting into various tree structures. After each insertion, test the structure of the tree with 3 different traversals. Test the to list method to make sure it returns a list representation corresponding to the in_order traversal. 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. It also shows that the balance function successfully does the rotation as desired so that the tree is balanced every time after insertion. This test provides a solid foundation for tests 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. Both double rotation and single rotations are included to make sure the remove function does not raise errors in various tree structures. 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. Test whether the to list function correctly returns the list representation of the AVL Tree. 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 balance function correctly rotates the tree so that 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 each time after insertion or removal.

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:



First insert 0, 1, 2, 3, 4, 5, 6 into the structure, then remove 3, 4, 5, 0 from the tree, so the structure is finally as above. This can ensure that the insertion and removal function correctly interact with each other and the balance function correctly rotates to form the desired balanced tree structure. In all, all the test cases work together to ensure that each method correctly interacts with the overall tree structure.

Part 3: Fraction explanation

In the Fraction class, there are mainly 3 methods implemented, less, greater, and equal. These three functions compare a certain fraction object self with another given fraction. The less method would return true if self is smaller than the given fraction and return false otherwise. The method here is to subtract the given fraction from self. If self is smaller, the resulting fraction would be smaller than 0 thus has opposite signs on numerator and denominator. So the product of denominator and numerator should be negative. Less would return true if the product is negative and return false otherwise. Similarly, if self is greater than the given value, subtracting that given value from self would result in a positive fraction. So the product of numerator and denominator should be positive. Greater would return true if product is positive and return false otherwise. If self is equal to that given value, subtracting that value would have a result of 0. So the numerator would be 0. Equal would return true under this circumstance. All these functions have constant time performance as it only involves arithmetics. Note that the subtraction function called in these methods also has constant time performance as it only applies arithmetics to the denominator and numerator of fractions. So the overall performances of the Less, Greater, and Equal methods are constant time.

Now consider the performance of the sorting using an AVL tree. As all the methods in the Fraction class only involve arithmetics, they all have constant-time performance. Even though to calculate the gcd of two value requires some iterations. Its relative performance is still relatively constant. Now consider use AVL Tree to sort n fraction objects. Creating n fraction objects would require linear-time O(n) performance as we have to insert two parameters to construct each fraction one by one. Inserting them into the AVL Tree would have an $O(n\log_2(n))$ performance as inserting a single element requires $O(\log_2(n))$ performance and there are n elements in total. So the overall performance of inserting n elements is $O(n\log_2(n))$. Then in order to get the sorted list, the to list function would be called. As explained above, this to list function has a linear time performance. So overall the performance is $O(n+n\log_2(n)+n)$. When n goes larger, the linear part would be neglectable. So the overall performance of this sorting method is $O(n\log_2(n))$. In the main section of my own code, I initially create a list containing integers from -1000 to 1000 which requires linear time. Then I use the shuffle function in the Random Module to randomize the order of elements in the list, this action also requires linear time. After that, I iterate through elements in the list to create a list of fraction objects with elements in the list as numerator and 120 as denominator. This action also requires linear time. So the performance of my own main section is actually $O(4n+n\log_2(n))$ as I use list comprehension to create objects to insert. If fraction objects are created and inserted one by one, the performance would be $O(2n+n\log_2(n))$ which is an overall $O(n\log_2(n))$ for large data sets.

In Project 1 when we implemented the insertion and selection sort, the worst-case performances of both methods are O(n^2), even though the code of these two sorting methods is much simpler than the AVL Tree version. It is clear that the AVL Tree sorting method really provides an edge over these two methods in sorting data. The complexity is worthy overall. Maybe, I will learn about some linear sorting methods in the future.