Question 1

The worst case performance of insert_element is logarithmic time because it calls the private recursive method __recur_insert which then calls the private __balance method, the private __rotate method, and the private __set_height method. The __balance method runs in constant time because it only carries out arithmetic operations to determine what rotation is necessary. Then, the __rotate method also runs in constant time because it only changes three of the parent-child links. The __set_height method runs in constant time as well because it only performs arithmetic operations to update the height of the particular node. Thus, the __recur_insert method runs in logarithmic time because it only calls constant time methods and since the __balance method ensures that the tree is balanced, the method will have to recur at most log(n) times in order to find the proper location for the new value. Therefore, the insert_method method also runs in logarithmic time.

The worst case performance of remove_element is logarithmic time because it calls the private recursive __recur_remove method which then calls the private __balance method, the private __rotate method, and the private __set_height method. As stated before, the __balance, __rotate, and __set_height methods run in constant time because they only carry out assignments and some arithmetic operations. The __recur_remove method itself runs in logarithmic time because it only calls the constant-time methods and it will need to recur at most log(n) times to find the value that needs to be removed. The method will only recur log(n) times because the __balance method ensures that the tree stays balanced and won’t trail off in any direction. Therefore, the remove_element method runs in logarithmic time since it calls the __recur_remove method.

The worst case performance of my implementation of to_list is quadratic time because it calls the private recursive method __recur_to_list which runs in quadratic time. The __recur_to_list method runs in quadratic time because it must recur n times in order to collect the value from every item in the tree. Then, within each recursion, the method appends each value to the list which takes linear time. The use of the linear time append method combined with the n recursions causes __recur_to_list to run in quadratic time. Therefore, the method to_list runs in quadratic time because it calls the quadratic time method __recur_to_list and otherwise only carries out constant-time assignments and conditionals. Although my implementation of to_list runs in quadratic time, there are other possible implementations that are able to run in linear time.

Question 2

To properly test the insert_element and remove_element methods, I first applied the same basic tests that I used in project 4 to ensure that the methods were able to correctly insert elements into and remove elements from the tree without needing to worry about rotations. Then, I moved on to testing cases that were more specific to the rotations required to keep the tree balanced. I tested insertions that caused single or double rotations in both directions. Then, I tested trees that had multiple insertions that required rotations to ensure that everything was being properly reassigned. To test remove_element, I first tested the three types of removals, with each type generating single and double rotations. Then, I tested a more complex removal case that resulted in two rotations from a single removal. Next, I tested removing multiple items from a single tree to ensure that the tree would be properly set up to remove more than one value. Lastly, I ensured that the removal and insertion methods work correctly with each other in one tree by inserting some values, then removing some, and then inserting more values while causing multiple rotations along the way. I checked all of the traversal methods and the get_height method in every case to ensure that the rotations were functioning properly both in the layout of the tree and in the height of the tree. I also added the to_list method to every test case that used the traversal methods, including the basic tests from project 4, to ensure that it correctly returned the in-order order of values and that its structure was correctly set up as a list of the values in the tree. To check the ladder part even further, I also inserted some strings into a BST to ensure that they were being correctly represented in the list.

To test the Fraction class comparison methods I inputted a wide range of fractions into a balanced tree. I made sure to input negative values into the numerator, denominator, and both to ensure that these values were being correctly compared to the other values in the tree. I also tested improper fractions to ensure that the fraction being greater than one wouldn't cause issues with the comparison operators. Then, I tested the comparison operators by inserting fractions with a zero in the numerator into a tree to make sure that they could properly handle a zero. Lastly, I tested my __eq__ method by using try/except blocks of code to ensure that the tree was correctly catching the ValueError of the same value being inserted more than once into the tree.

Question 3

To use this sorting method, you first must insert all of the items into the balanced BST and then obtain either the to_list or in_order traversal of the tree. As stated before, the insert_element method runs in logarithmic time for each insertion, so inserting n items into a balanced BST will take n*log(n) time. Then, if implemented efficiently, both the in_order and to_list traversals can operate in linear time. These two runtimes are added together, not multiplied because they are two separate operations. Therefore, this sorting method will run in n*log(n) time, which is an improvement over the quadratic time performance that is found when sorting with an unbalanced BST.

This sorting performance will not change depending on the types of objects because the structure of the sorting method remains the same. As long as the object has less than and greater than comparisons, the type of object inserted into the tree doesn't impact the logarithmic insertion time. The type of object also doesn't impact the runtime of the traversal methods, so the sorting performance will remain at n*log(n) time.