Question 1

First, the get_height method runs in constant time because it only accesses the height value that is stored in the root node. The value of the height is changed during the return of the insertion and removal recursive functions, allowing get_height to run in constant time. Next, the worst-case performance of the insert_element method is linear time because it calls the private recursive method __recur_insert, which runs in linear time. The method can not guarantee that the tree will be balanced, so at most, the recursive method will have to recur n times to find the correct place for the new value. All of the subsequent linking and height operations in the recursive method occur in constant time, so insert_element runs in linear time. With unbalanced trees allowed, the method remove_element also runs in linear time because it calls the private recursive method __recur_remove, which will need to recur at most n times in order to find the value that needs to be removed. Then, all of the following linking and height operations in __recur_remove occur in constant time, so remove_element is able to run in linear time. This runtime applies even when removing a node that has two children because the second recursive call to __recur_remove is called on the subtree rooted at the removed node's child, so no more than n recursive calls will be made. Finally, the traversal methods in_order, pre_order, and post_order all run in quadratic time because of my use of string concatenation to form the output strings in their respective private recursive functions. By convention, the traversal methods all have to visit every node in the binary tree. So, their recursive methods are called n times, and with the recursive methods performing a string concatenation with each node, the traversal methods run in quadratic time. Finally, the conditional statement, slice, and the one additional concatenation present in each non-recursive traversal method do not impact the quadratic runtime caused by the calls to the recursive methods.

Question 2

I’ve ensured that my methods work in all cases by testing them all on empty trees, on balanced trees, and on unbalanced trees. I know that the methods will be successful when the tree is large because the use of recursive functions makes it so that each node is treated as its own subtree, so larger trees won’t change the way the functions operate. I know that the operations will work correctly when the tree is tilted because I created test cases for this possibility and because an unbalanced tree shouldn’t cause problems in the recursions for insertions and removals. For unbalanced trees, I also made sure to test my get_height function to check that the height was being correctly computed with the height of the node’s tallest child. I believe I have tested all possible scenarios for insertions and removals. I tested the insert_element method with just one element, then with inserting one value to the left of the root, and then by inserting one value to the right of the root. Next, I checked that insert_element functioned correctly when inserting elements into a balanced tree, into a tree tilted to the left, and into a tree tilted to the right. Then, to finish testing insert_element, I tested several different cases where I tried to insert a value that was already in the tree to ensure that the method correctly raised a ValueError and didn’t alter the structure or the height of the tree. For the method remove_element, I tested the removal of a root node with no children, with one child, and with two children. Then, I tested the removal of a leaf node from a tree with a height greater than one, the removal of a node in the middle of a tree with one child, and the removal of a node in the middle of a tree with two children. Next, I tested the remove_element method on an empty tree and with values that were not in a tree to ensure that the method correctly raised a ValueError and didn’t alter the height or structure of the tree. Lastly, I included a test case where I inserted values into a tree, removed some values, and then inserted more values to ensure that the two methods work properly when used after each other.

Question 3

As binary search trees rely on the principle that a node's left child is less than itself and a node's right child is greater than itself, any object that can utilize the less than and greater than operators can be inserted into a binary search tree. Therefore, float objects and even strings can be inserted into a binary search tree. Inserting strings into a binary search tree will sort them based on alphabetical order so that strings to the left of the root are all alphabetically before the root and strings to the right of the root all appear later on in the alphabet.

In order to obtain a string representation of the values in increasing order, you can use the in_order traversal method. This traversal sorts the values because it first traverses the left child, which is smaller than the parent node, then the parent node, and lastly the right child, which is greater than the parent node. The performance of this sorting approach is $O(n^2)$ because in this implementation, the binary tree is not guaranteed to be balanced. Therefore, inserting n elements into the tree, where each insertion takes linear time, results in a quadratic runtime. Additionally, with my traversal methods, the in_order traversal of the newly created tree will also result in a quadratic runtime.