Anatoly Osgood
11/17/19

BST Class

The first piece of my BST file is the creation of the Node class. Inside of it, the attributes value, left, right, and height store values of the Node. The Node class is passed a value on creation which becomes its value. Left and right are initialized as None until later if the node has children. Finally, the height is initialized at 1 because, as discussed in class, a node without any children is of height 1. The worst-case performance analysis of this method is constant time.

Moving onto the init method for the BST class, I create a private variable representing the root of the tree. I set it equal to None initially as there is no tree nor a node yet. The worst-case performance analysis of this method is constant time.

The public method insert_element takes a value that is to be inserted into the tree. It calls the private method __insert_element as per the requirements of the spec and sets the outcome equal to the trees root variable. The private method takes the value the user desires to be inserted along with the root of the entire tree. This is because the node is initially inserted at the root of the entire tree but then navigates to the correct point of insertion as I will describe in the next paragraph. The worst-case performance analysis of this method is linear time. This is because the performance of this method is dependent on the performance of the insert_element method.

The private insert_element method is a recursive method as per the spec. The first two if conditions are base cases. The first if statement creates a node at that subroot of the subtree if it is None using the value passed by the user. If the method navigates to where the node should be inserted but a node of that value is already present, the method raises a ValueError as per the spec. Otherwise, the method compares the value to be inserted at the current root that the method is at. As the rules of a binary search tree requires, if the value to be inserted is less than the value at the current root of the tree, the method calls the value to be inserted at the child to the left (root.left) and sets the left child equal to that outcome, starting another recursive process. If the value to be inserted is greater than the value at the current root of the tree, the method calls the value to be inserted at the child to the right (root.right) and sets the right child equal to that outcome, starting another recursive process. The recursive process continues until the method

hits one of the previously mentioned base cases (the first two if statements of the method). After the recursive call to either of the root's children, the method also makes a recursive call to set the height of that root. In either recursive case, the height of the root is set equal to the value retrieved from calling the private height_checker method on itself. I will describe how this method finds the height of each subroot traversed below. Finally, this method returns the value of the root created so that root of the entire tree class is set equal to this value. This allows for the tree to reformed with the inserted node. The worst-case performance analysis of this method is linear time. This is because the worst-case of this unbalanced binary tree is that all the nodes go either towards to the left or right of the root node. Then, this method would have to walk through every single node in the tree through the recursive calls in the method.

The public method remove_element takes a value that is to be removed from the tree. It calls the private method __remove_element as per the requirements of the spec and sets the outcome equal to the trees root variable. The private method takes the value the user desires to be removed along with the root of the entire tree. This is because the node to be removed is somewhere in the tree and the method must traverse the tree to find the node.

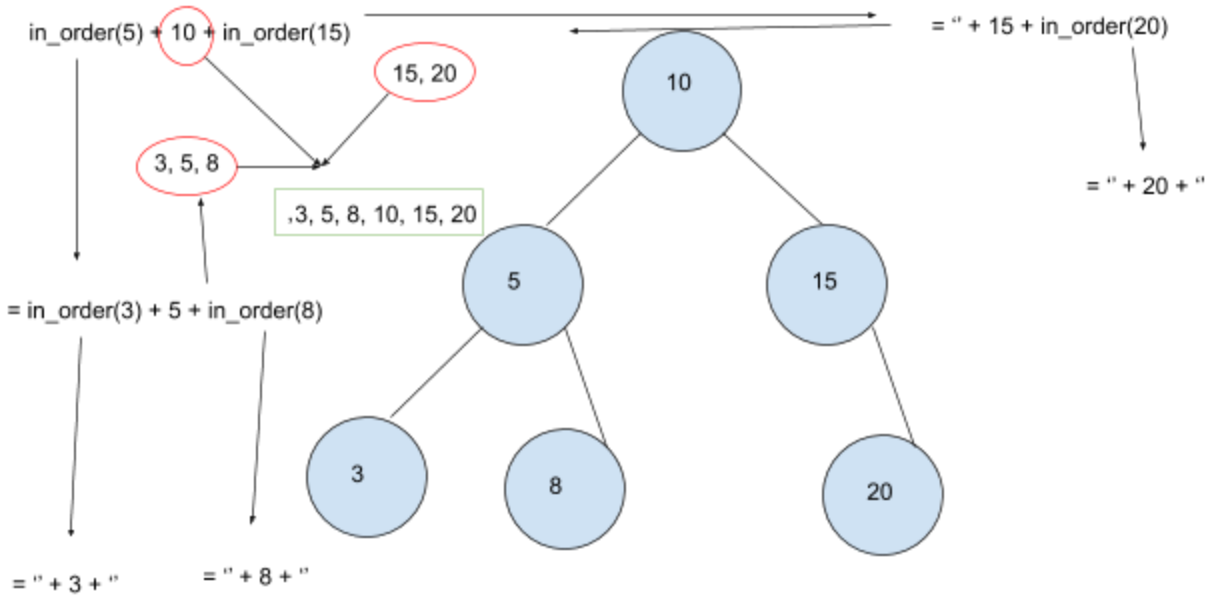
The private method remove_element takes the value passed by the user and begins its search for the node to remove at the root of the entire tree. The first base case is if the root is None, then a ValueError is raised as there is no tree and nothing to remove. The next base case is if the root value is equal to the value that is to be removed. If that root has no children, then the method returns None in order to remove that node. If that root has a child, (either left child is None and right child isn't None OR left child isn't None and right child is None) then the method returns the child that isn't None. If the root has two children, then the method enters the right subtree of the root and walks though the left side of the subtree using a while loop until it hits None. This is done to find the smallest value in the right subtree of the root. Then I replace the value of the root with this smallest value as per the spec. Finally, I remove this smallest value of the right subtree by recursively calling the private remove_element method providing it the smallest value of the right subtree and the root of the right subtree. Just like the private insert_element method, I then call the private height_checker method to set the new value of the height of that specific root after the remove_element operation. Finally, I return the root value.

The other two if statements are essential for walking the tree in order to find the nodes that are to be removed. As the rules of a binary search tree requires, if the value to be removed is less than the value at the current root of the tree, the method calls the value to be removed at the child to the left (root.left) and sets the left child equal to that outcome, starting another recursive process. If the value to be removed is greater than the value at the current root of the tree, the method calls the value to be removed at the child to the right (root.right) and sets the right child equal to that outcome, starting another recursive process. I again call the private height_checker method to set the new value of the height of that specific root after the remove_element operation. Finally, I return the root value in order to set the new value of the private root variable of the entire tree. The worst-case performance analysis of this method is linear time. This is because the worst-case of this unbalanced binary tree is that all the nodes go either towards to the left or right of the root node. Then, this method would have to walk through every single node in the tree through the recursive calls in the method.

The next method is the pubic in_order string method. This method concatenates the opening and closing brackets onto the string returned from the private in_order method. It provides the private in_order method the root of the entire tree as this method is called for one of the three different ways of representing a binary search tree. This method also slices the first two indices off the string returned because otherwise the string would have an extra space and comma due to how the private in_order method operates. I will explain this in the next paragraph. The worst-case performance of this method is linear time as this method is entirely dependent on the performance of the private in_order method which performs in linear time. I will explain why the private in_order method performs in linear time in the next paragraph.

The private in_order method takes a value of the root that is to be displayed. If the root is None and there is no tree it returns nothing as there is nothing in the tree. Otherwise, it recursively calls itself for the left child of the root, adds a comma, adds the string value of the value of the root, and recursively calls itself for the right child of the root. It concatenates all of these values into a string called tree which is the string representation of the tree. Finally, the method returns tree which allows the representation of the tree to be displayed. The reason that this method calls the left child, root, and then right child is because in-order traversal first

traverses the left child, root, and then right child. Also, recursion allows us to traverse the entire tree and represent it through strings. By recursively calling the left child, the method creates all the values of the left subtree concatenated together, adds the root value, and then does the same thing for the right subtree. For example, if the method is calling in_order(root.left) and hits a leaf node, the method would simply return the node's value as the left and right children are None. This simple but elegant solution allows the method to "walk" all the way down the tree and then slowly "climb" back up through recursion, concatenating nodes as the method walks back up to the root of the entire tree. The reason that the public method has to slice the first two indices off of the string value returned by the private in_order method is that the way I created the method there is always an extra space and comma due to the fact that eventually the method hits the case where the left child is None. The drawing below illustrates how the in_order method creates a representation of the tree (note the extra comma and space the front - this is shaved off in the public method when we slice the first two values off).



The performance of this method is linear time as every single value of the tree is touched.

The next method in the file is the public pre_order method. This method is identical to the public in_order method except that it calls the private pre_order method but also provides it the private root variable that represents the root of the entire tree. It also adds the opening and closing brackets to the string value returned from the private pre_order method, along with slicing the first two indices off what is returned. The worst-case performance of this method is linear time as this method is entirely dependent on the performance of the private pre_order method which performs in linear time. I will explain why the private pre_order method performs in linear time in the next paragraph.

The private pre_order method takes a value of the root that is to be displayed. If the root is None and there is no tree it returns nothing as there is nothing in the tree. Otherwise, it adds a comma, adds the string value of the value of the root, recursively calls itself for the left child of the root, and recursively calls itself for the right child of the root. It concatenates all of these values into a string called tree which is the string representation of the tree. Finally, the method returns tree which allows the representation of the tree to be displayed. The reason that this method calls the root, left child, and then right child is because in-order traversal first traverses the root, left child, and then right child. This method works identically to the private in_order method but simply does it in a different order because of the difference between inorder and preorder traversal. The performance of this method is linear time as every single value of the tree is touched.

The next method in the file is the public post_order method. This method is identical to the public in_order and pre_order methods except that it calls the private post_order method but also provides it the private root variable that represents the root of the entire tree. It also adds the opening and closing brackets to the string value returned from the private post_order method, along with slicing the first two indices off what is returned. The worst-case performance of this method is linear time as this method is entirely dependent on the performance of the private post_order method which performs in linear time. I will explain why the private post_order method performs in linear time in the next paragraph.

The private post_order method takes a value of the root that is to be displayed. If the root is None and there is no tree it returns nothing as there is nothing in the tree. Otherwise it

recursively calls itself for the left child of the root, recursively calls itself for the right child of the root, adds a coma, and calls the string value of the root. It concatenates all of these values into a string called tree which is the string representation of the tree. Finally, the method returns tree which allows the representation of the tree to be displayed. The reason that this method calls the left child, right child, and then root is because post-order traversal first traverses left child, right child, and then root. This method works identically to the private in_order and pre_order methods but simply does it in a different order because of the difference between inorder, postorder, and preorder traversal. The performance of this method is linear time as every single value of the tree is touched.

The get_height method returns the height of the entire tree by returning the value of the root's height. If there is no tree and no root (root is None), then the height is zero, so the method returns 0. Otherwise, it returns the root's height. The root's height is kept accurate through the use of the private height_checker method. I will explain how this method works below. The worst-case performance of this method is constant time.

The str method allows the user to print the string representation of their tree. If there is no tree so the root is None, the method returns empty brackets as per the spec. Otherwise, it calls either the in_order, pre_order, or post_order function to get the representation of the string. The worst-case performance of this method is linear time as it would be when one of the three traversal methods gets called which all operate in linear time.

The final method is the height_checker method. This method takes a root value and gives the height of the input root. I used this method to re-calculate the height of all nodes impacted by insertion or removals in the private insert_elemennt and remove_element methods. Right after the recursive call, this method was accessed allowing me to set the root's height equal to the output of this method. This is key to keeping the performance of get_height in constant time. If the root given is None, then the height returned is zero. If the root give has no children, then the root has a height of 1. If the root only has a right child, then the method returns the value of the child's height plus one as the root is one level higher. On the other hand, if the root only has left child, then the method returns the value of the child's height plus one as the root is one level

higher. If the root has both a left and right child, then the method returns the maximum of either child's height plus one. The performance of this method is constant time.

Testing

The first thing I did was create my tree for testing. Then I tested the empty tree's representation and height were correct as these are important edge cases. I also tried to remove the root value from the empty tree which correctly threw a ValueError. I tested inserting values into the tree and making sure that the string representation was correct. I also tested the height was correct with just the root, that the height was the same when the root had no, one, or two children, and that the height correctly increased when a fourth node was added. I also tested that the method threw a ValueError when a duplicate value was attempted to be inserted. I also tested that if the tree had a ton of nodes on its left side or right side, the get_height method correctly worked and returned the larger height. I then tested string representations for one through four and seven values for the inorder, preorder, and postorder representations of the tree. I did this by changing which method was called in the str method and by commenting out the other two types of traversals while testing. Finally, I tested the removal method and made sure the string representation of the tree was correct when removing subroots and the root.