Anatoly Osgood
12/4/19
AVL Tree Writeup

Fixes to BST

Before I talk about the additions I made to my BST file (balance method), I want to talk about a few changes I made to my existing BST code. As I was creating my AVL tree, I realized that my height_checker method (called to adjust the height on each node when insert_element or remove_element was used), had a small error. Instead of using elifs, I used an if clause on each condition. When trying to create the string representation of my tree, I kept hitting maximum recursive depth and wasn't sure why. It turned out that because I hadn't used any elifs, my code kept walking through the height_checker if-statements trying to find the applicable one when constructing the string representation of my tree, causing the maximum recursive depth error. As a result, I switched all clauses to elifs and made sure to consider their use in my balance method. This change doesn't change the performance of this method.

Additions to BST

insert and remove

I added calls to the balance method at the end of each method. This is to re-balance the tree upon completion. This does not change the performance of either method as the balance method operates in constant time.

balance

The first change I made was the creation of the balance method which completes the required rotations to ensure the tree created is an AVL tree. I used the pseudocode on Blackboard to guide my approach in the creation of the method. It is important to mention that I also added two key methods from the textbook. I struggled at first to consider how to handle if the children of the node are None. The textbook had two methods which were very helpful in handling this case. The methods are called __left_child_checker and __right_child_checker. Both of these methods are in constant time. These methods take a node and examine the nodes left or right child (depending on which method is called). If there is no child node, then the method returns a height of 0, otherwise it returns the height of the child. These methods are key for identifying imbalance and guiding how the balance method works.

Moving onto the balance method, it takes a root value t which is the node that is imbalance is calculated across. The first clause considers whether the node that is passed even exists. If the node does not exist (is None), then the method simply returns the node passed to the method. Otherwise, if the node passed is left-heavy by 2, the method chooses one of two ways of rotating about the node. A node is calculated to be left-heavy by 2 by comparing the heights of the node's right and left children. If the height of the right child minus the height of the left-child is negative 2, then that means the node is left-heavy by 2. Then, there are 2 subclauses for the 2 different types of rotations. The first clause checks if the node's left child is left-heavy by 1 or balanced by 0. I check this by using the two methods from the textbook by finding the height of the right child of the left child of the node and subtracting the height of the left child of the left child of the node. If this value is -1 or 0, then this clause is fulfilled. This requires a single rotation right about the node. To complete this rotation, I first call a variable called holder which points at t. Then I set t to the left child. In case, there is a floater variable I also call holder's new left child to point at t's right child. Finally, I set the new t's right child to be holder (the old t). This completes the rotation. I didn't change the new t's left child value as that remains constant in this rotation case. I then set holder's height to whatever is returned by calling my height_checker method on holder. This to update holder's height as I then call the height_checker method on the new t, updating its height.

The other clause in the case the node t is left-heavy by 2 is when the left child of t is right-heavy by 1. This is calculated by finding the height of the right child of the left child of t and subtracting the height of the left child of the left child of t. If the difference is 1 or the first subclause wasn't true, the method enters this clause. Per the rules of the AVL tree, this requires two rotations. The first is a left rotation about t's left child and the second is a right rotation about t. The first left rotation is accomplished as follows. First, I defined holder and left_child to point at the left child of t. Then, I created the variable left_floater to point at t's left child's right child's potential left floater. I set the new value of t's left child to point at the original right child of the left child of t. Then I set t's new left child's left child to be holder. Then, to account for a floater, I set holder's right child equal to the variable left_floater. I then set holder's height to whatever is returned by calling my height_checker method on holder. This to update holder's height as I then call the height_checker method on the new t.left, updating its height. Finally, I update t's height by calling the height_checker method on itself. Then I completed the same right rotation about the node t I did in the previous clause and updated the heights.

Otherwise, if the node passed is right-heavy by 2, the method chooses one of two ways of rotating about the node. A node is calculated to be right-heavy by 2 by comparing the heights of the node's right and left children. If the height of the right child minus the height of the left-child is 2, then that means the node is right-heavy by 2. Then, there are 2 subclauses for the 2 different types of rotations. The first clause checks if the node's left child is right-heavy by 1 or balanced by 0. I check this by using the two methods from the textbook by finding the height of the right child of the right child of the node and subtracting the height of the left child of the right child of the node. If this value is -1 or 0, then this clause is fulfilled. This requires a single rotation right about the node. To complete this rotation, I first call a variable called holder which points at t. Then I set t to the right child. In case, there is a floater variable I also call holder's new right child to point at t's left child. Finally, I set the new t's left child to be holder (the old t). This completes the rotation. I didn't change the new t's right child value as that remains constant in this rotation case. I then set holder's height to whatever is returned by calling my height_checker method on holder. This to update holder's height as I then call the height_checker method on the new t, updating its height.

The other clause in the case the node t is right-heavy by 2 is when the right child of t is left-heavy by 1. This is calculated by finding the height of the right child of the right child of t and subtracting the height of the left child of the right child of t. If the difference is 1 or the first subclause wasn't true, the method enters this clause. Per the rules of the AVL tree, this requires two rotations. The first is a right rotation about t's right child and the second is a left rotation about t. The first right rotation is accomplished as follows. First, I defined holder and right_child to point at the right child of t. Then, I created the variable right_floater to point at t's right child's left child's potential right floater. I set the new value of t's right child to point at the original left child of the right child of t. Then I set t's new right child's right child to be holder. Then, to account for a floater, I set holder's left child equal to the variable right_floater. I then set holder's height to whatever is returned by calling my height_checker method on holder. This to update holder's height as I then call the height_checker method on the new t.right, updating its height. Finally, I update t's height by calling the height_checker method on itself. Then I completed the same left rotation about the node t I did in the previous clause and updated the heights.

The worst-case performance of this method is constant time as specified in the project specification. This is because every clause rotates nodes similar to the way that we swapped nodes in the linked list data structure (the same way every time). There is no dependence on the amount of nodes in the tree and the rotations of nodes work the same way every time. Also

to_list

My public to_list method represents the tree as a list as specified by the project requirements. It calls the private __list__ method which walks through the tree in the in_order pattern. It passes the private __list__ method the root of the entire tree which is required to traverse the nodes of the tree. The worst-case performance of this method is linear time as it calls the __list__ method which operates in linear time. I will explain why __list__ operates in linear time below.

If the root passed to __list__ is None and there is no tree, then the method simply returns an empty array as there is nothing in the tree. Otherwise, it calls itself on the left child of the root, concatenates the value of the root (adding brackets around it so that it can be added to the list variable tree_list which holds the list representation of the tree. Finally, it concatenates the output of calling itself on the root's right child. This method works exactly like the in_order method, but instead of casting the root value as a string I put brackets around it so that it could be pushed in the list variable tree_list. Finally, it returns tree_list which is returned by to_list. The performance of this method is linear time as the method goes through every single node of the tree.

Fraction file

lt, gt, and eq methods

The first thing I did in my Fraction file was construct the private __lt__, __gt__, and __eq__ methods which check whether two fractions are less than, greater than, or equal to each other respectively. The key to these methods was to figure out how to compare any two fractions. Two fractions can be compared if you multiply each fractions' numerators and denominators by the other fractions denominator. This makes both fractions' denominators identical, so you can compare the fractions' size. However, you don't need to manipulate the denominators as only the numerators matter.

For the __lt__ method, I created self_num which is the first fraction's new numerator. It is the product of the numerator and the other fraction's denominator. I created other_num which is the other fraction's new numerator. It is the product of the numerator and the first fraction's denominator. Then, I compared the self_num and other_num. If self_num is less than other_num then the method returns True as the first fraction is less than the other fraction. Otherwise, it returns false. The performance of this method is constant time as it's performance has no dependence on the size of its input (a fraction object).

For the __gt__ method, I created self_num which is the first fraction's new numerator. It is the product of the numerator and the other fraction's denominator. I created other_num which is the other fraction's new numerator. It is the product of the numerator and the first fraction's denominator. Then, I compared the self_num and other_num. If self_num is greater than other_num then the method returns True as the first fraction is greater than the other fraction. Otherwise, it returns false. The performance of this method is constant time as it's performance has no dependence on the size of its input (a fraction object).

For the __eq__ method, I created self_num which is the first fraction's new numerator. It is the product of the numerator and the other fraction's denominator. I created other_num which is the other fraction's new numerator. It is the product of the numerator and the first fraction's denominator. Then, I compared the self_num and other_num. If self_num is equal to other_num then the method returns True as the first fraction is equal to the other fraction. Otherwise, it returns false. The performance of this method is constant time as it's performance has no dependence on the size of its input (a fraction object).

Sorting fractions

In the main method, I tested sorting of Fraction objects using the AVL tree implementation as required by the project specification. First, I created a bunch of Fraction objects in a random order. Then I inserted them into an array and printed it to see the unsorted values. Then I created an AVL tree called test and inserted the objects into the tree. Finally, I printed the in_order representation of the tree using the to_list method to see if the values were correctly sorted. The performance of the main method is dependent on the insert_element method. We are using an AVL tree though, so the performance of a single insertion is log time, but with n values is n log n time. The to_list method operates in linear time as mentioned above, but we only care about the worst-case performance which is based on the insert_element method. Therefore, the worst-case performance of the Fraction main method is n log n.

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. Finally, I tested the removal method and made sure the string representation of the tree was correct when removing subroots and the root.

I also watched to make sure my new additions were working correctly. So I tested each rotation, checking the list and string representations, along with the height was working as expected.

I also made a larger and complicated AVL to check that all rotations were working correctly and the height continued to work. I then removed different nodes to make sure remove_elemennt was working as required and the height was changing when necessary.