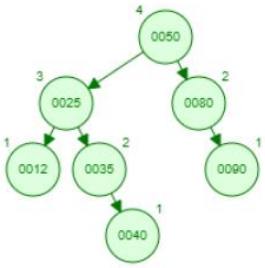
Project 5: AVL Trees

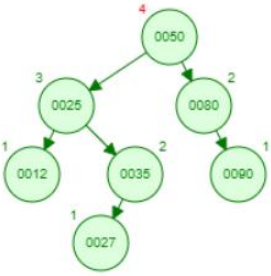
Project 5 aims to further knowledge of binary search trees by continuing adding a balance element to Project 4. This project includes three files: Binary_Search_Tree.py, BST_Test.py, and Fraction.py.

Binary_Search_Tree.py

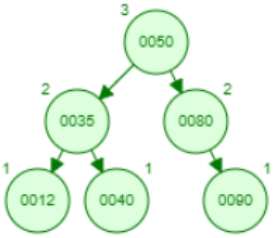
Updates to the Unbalanced Binary Search Tree Algorithm

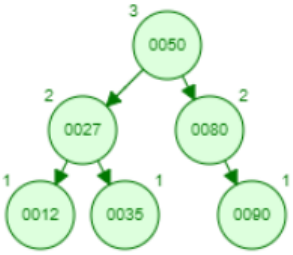
Although feedback was not returned for the original unbalanced tree, an error was found while arbitrarily testing trees. Unfortunately, this error was not captured in the unbalanced test file due to it only affecting few cases. While testing one of my balanced trees, I realized the height would not always correctly update when a node with two children was removed. Although it did work for the removal in my test case, it did not work for the combination I put in arbitrarily. Below are the arbitrary trees I created:





I next tried to remove node 25. In both examples, the trees' updated heights were still four. However, as we see below, the trees' heights should have been three.

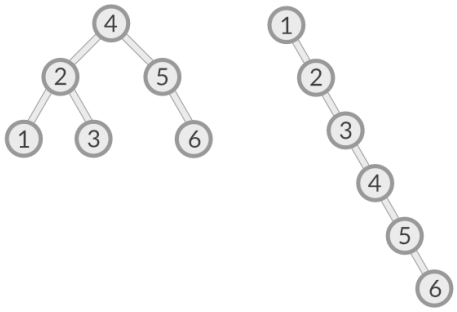




This error was very intriguing, as all my other removals functioned correctly. If I added more nodes to the left sides of the trees, the height would correct itself. In order to find my error, I inserted print statements in my height function method and my removal recursions. With print statements, I was able to see what my code exactly did and where it missed a spot. I noticed that my height function would skip over the node that took the place of 25, meaning that I had forgotten to update the height somewhere. By inserting some more print statements within the third case of my removal function, I realized I needed to add an update height call before returning t. This fixed my error.

From Unbalanced to Balanced

Balanced Binary Search Trees are BSTs which update their placement in order to comply by certain balancing rules. These balancing rules state that the heights of the left and right sides of each subtree in a BST cannot differ by more than one level. In the picture, we can see that the left balanced binary tree has relatively symmetrical density, while the right tree is very skewed towards the right. What does this mean for our algorithm? A balanced binary search tree will insert or remove a node in the same way an unbalanced BST does, but then will



update the structure to abide by the balancing rules. Therefore, we can continue off from Project 4. When inserting or removing a node, a private recursive method was called which would return the root of the current subtree. For balanced trees, we must insert one more step into the process. After completing a recursive call, rather than returning t, we return a balanced t. A private __balance method was created to delineate how the tree needs to be restructured depending on its case. The first case occurs when t is None. However, unless there is an error within the insertion or removal methods, t should not be None, as it was addressed in earlier steps of the process. Our second outlier case that must be addressed in order to avoid an attribute error is the case in which t does not have children. Without these lines, Python would look at the next case and find that the children do not have height attributes since they do not exist. Therefore, single node insertion must be addressed before the rest of the cases. The third and fourth cases address unbalanced trees. If the tree is skewed towards the left, it requires a right rotation. In order to determine whether it is unbalanced or not, either the balance factor needs to be calculated or the right side of the tree needs to be unpopulated(as is a descending linked list). Within the left leaning cases, there are two subcases. If a tree is skewed left and its biggest subtree is also skewed left, one single rotation to the right occurs. If the subtree is skewed right but the tree is skewed left, a double rotation occurs. The four right leaning cases follow the same logic in reverse. In order to find these cases and subcases, several conditionals with many and/or statements were coded. These conditionals cover each circumstance. Although this is functional, it may be a little bit cluttered and hard to read. If I were coding this again in the future, I would consider creating another method which would contain these conditionals, especially since some aspects of the conditionals are repetitive. If I did not want to make another method, I could have also expanded __balance by assigning variables to parts of the conditionals. For example, t.left.right.height-t.left.left.height was continuously used throughout the conditionals. Perhaps this could have been taken care of with a variable assignment beforehand.

The balance method calls upon the left and right rotation functions depending on the case. These two functions are essentially identical but done in reverse. Both the right rotation and left rotation methods are done in constant time since we are just reassigning values/pointers and updating the height. We begin both methods by storing the initial subtree values including the right and left children of t, as well as some of the grandchildren. This was done so values are not lost throughout the rotation. In order to rotate the nodes, we need to establish a new root node and make sure that the left and right attributes are pointing at the correct nodes. Once this is completed, the heights need to be updated. I struggled a lot with this concept at first since I did not realize that I could put in any node into my height function. After drawing dozens of trees, I started seeing a pattern. Rather than going down through the entire tree and updating all of the nodes' heights, we only needed to update the heights that had changed due to the reassignment. In every case, the old and the new root (t) change heights. The children only changed height if they existed. In order to avoid running into attribute errors (None can't have a height attribute), I decided to place conditionals before updating their heights. I suppose I also could have avoided this by adding some lines to my height function to overpass any None values. This may have optimized the balance function and my rotation functions if implemented successfully. Another error that I battled with was returning the updated root value to the balance function. I realized that in the balance function, the rotation function needed to update the new t value, and thus I had to return the new root in the rotation functions. The balance function is essentially doing inspecting the current structure of the tree and then sending it off to the rotation functions. Because the rotation functions are constant time, the balance function is constant time. Because the balance method returns to the private

insertion or removal methods, we need to consider whether there are any changes to the runtimes. Because the balance and rotation functions are constant time, they have no impact on the runtimes of any other methods. Thus, the runtimes for all the project four methods are somewhat the same as previously specified. However, it is important to note that the worst case scenario of the unbalanced BST is different than that of the balanced BST. Whereas before the worst case for insertion or removal was traversing to the end of a linked list, now we have assurance that the distribution of nodes is more even. Thus, we will need to travel through less nodes. Our balanced BST insertion and removals will now be O(log n) runtime.

The only other edit made to the Binary_Search_Tree file was to include a new output method. The to_list method returns the binary search tree in sorted order as a list. Because the inorder traversal sorts BSTs, the elements from the tree are appended to the list using this traversal method. In Project 4, my implementation of the three traversals was done with a list rather than string concatenation. Therefore, there were very few things that needed to be changed from the original inorder implementation. I did not need a private to_list method because the private inorder method was already coded with the append function, and thus the to_list method could use it. The only changes that were made were removing the string elements. The list could be returned to the user with no changes. To_list will have the same runtime as the inorder function, as it depends on its private method.

BST_Test.py

The BST_Test.py code includes many more test cases than in Project 4. I discovered on piazza that more than one assertEqual function could be called per test case, and thus I was able to reorganize a lot of the cases. The cases are ordered by what they are testing. I tested insertion, removal, height, and right and left rotations. Before adding the rotation test cases, I chose to keep a lot of the insertion and removal test cases even if they were repetitive (rotation is used in insertion and removal, so testing the functionality could have been done with less test cases). The insertion, removal, and height functions all needed to be updated since the ordering of the tree is different in a binary search tree. In order to try to separate recursion and rotation testing, I did my best to insert values into the tree equally to avoid triggering the rotation function. Although I cannot say this is the case for all the insertion functions, there are enough cases which isolated recursion. Had I had any errors in my recursion, the split between recursion and rotation would have helped me problem solve more easily. This is exactly how I discovered that my removal recursive function was missing an element. Not much else was changed in the insertion, removal, and height sections.

My rotation test cases were a little bit difficult to implement, as I had to go through each conditional in the balance method and make sure all the occurring combinations were tested. Left and right rotation test cases follow the same logic and are essentially the same in reverse. Although I drew many trees and tried to cover as many combinations as I could, I am not confident that I covered all of them. Because of my doubts, it was very important to insert a multitude of random trees into the file and see if any problems would occur. This was done throughout my insertion, deletion, and height test cases as well as off the test file for practicality purposes. Although I covered several conditions in both the right and left rotation test cases, I would have liked to have more test cases in which more than one rotation (separate from double rotations) was performed. The last removal test case does exactly this; it inserts many nodes and has multiple balancing functions. Since there are at least 40 left and right rotation test

cases, this would have added over three hundred more test cases. For the sake of space and time, it is more efficient to randomly select combinations as is done in the last removal test case.

Lastly, to_list was tested with the assertEqual function as was done with the traversal methods. I was surprised to see that an empty list returned a string '[ ]'. These were the only errors within my test cases (which were fixed promptly).

Fraction.py

The Fraction.py file served as a method of handling inserted fractions into the BST. At first, I was very confused by the functionality of this file, as I thought Python would preserve the numerator and denominator when inserted. However, Python seems to convert everything into float objects. This file is able to overcome that tendency. The Fraction class initializes the numerator and denominator attributes and calls a reduce method. The reduce method reduces the fraction by finding the absolute value of the inserted numerators and denominators and divides both by the GCD (which was also included as a static method). I am largely unfamiliar with static methods but was interested in seeing that it doesn't require an object. I am assuming that this may have been the reason it was coded like this, as it was implemented when the Fraction class was called. The next four methods add, subtract, multiply, and divide the fractions with manipulations of the numerator and denominator. The lt, gt, and, eq methods are crucial to the BST. We know that fractions are division and that Python will turn fraction into float objects. We also know that Python can compare floats. This logic was used to implement these three methods. For example, the lt method introduces two objects which have numerator and denominator attributes. By dividing these two attributes and then comparing them with a < sign, we can determine whether or not the first fraction is less than the other. If it is, the method returns true. If not, it will return false. This is applied throughout the other two methods with their respective comparative signs. Another way to implement these methods was through multiplication. However, all other approaches are less readable and straightforward than the one taken. It is also important to note that the reason Boolean values were returned is because of the place the Fraction class is called. In order to observe this pathway, I inserted a non fraction value into our BST with other fractions. Analyzing the error below can tell us more about how the Fraction class works.

```
AttributeError                            Traceback (most recent call last)
<ipython-input-14-e003bde5e3d9> in <module>
    102   array=[0,Fraction(12,4), Fraction(1,9), Fraction(1,100), Fraction(3,20), Fraction(4,9), Fra
ction(10,9), Fraction(11,9),Fraction(1,200), Fraction(1,300), Fraction(1,500), Fraction(1,100000), Fr
action(-1,2)]
    103   for i in array:
--> 104     BST.insert_element(i)
    105   print(BST.to_list())
    106   print('Unsorted Array:'+ str(array))

<ipython-input-12-cdc9d173bf41> in insert_element(self, value)
     14   def insert_element(self, value):
     15     #Private Function called with Root as parameters
---> 16     self.__root=self.__rins(value, self.__root)
     17     return self.__root
     18

<ipython-input-12-cdc9d173bf41> in __rins(self, x, t)
     27         t.height=1
     28     #Recurring to the Left/Height Function is called
---> 29     elif x< t.value:
     30         t.left=self.__rins(x,t.left)
     31         self.height_function(t)

<ipython-input-14-e003bde5e3d9> in __lt__(self, other)
     51
     52   def __lt__(self, other):
---> 53     if (self.__n/self.__d)< (other.__n/other.__d):
     54         return True
     55     else:
```

We note that the value first passes through the public insert function and then to the recursive private function. As the code encounters the first less than < sign, it immediately looks at the Fraction methods. When it calls up the respective methods, it will return True or False to the conditional in the recursive insertion class. If true, that conditional will proceed. The fraction methods are all constant time, as they all deal with a known attribute of an object and apply some mathematical property to them. Python only needs to call the object and its attributes and perform comparisons and operations. Thus, it has no effect on the runtime of the general BST. The main class of Fraction.py inserts both negative and positive fractions into an array. In order to make it more readable, variables were assigned to each inserted object in the list. This makes it much easier to change the array. In order to insert them into the BST at once, a for loop was created to run through the list. By printing the to_list method in the BST file, we get a sorted array!

Works Cited

Agarwal, userMeenakshi. “Python Static Method Usage Explained with Examples.” *Learn Programming and Software Testing*, 18 Mar. 2019, www.techbeamers.com/python-static-method/.

Visualization used: https://people.ok.ubc.ca/ylucet/DS/AVLtree.html

Picture Used: https://i.stack.imgur.com/tjkrr.png