Project 4: It's Just a Jump to the Left and a Step to the Right

Project four aims to implement recursive algorithms and unit testing to an unbalanced binary search tree. This project includes two python files: Binary_Search_Tree.py and BST_Test.py.

Binary_Search_Tree.py- the Binary_Search_Tree class

The Binary_Search_Tree file initializes the binary tree, allows functions of insertion and deletion, and includes three types of traversal methods.

Initialization of Nodes:

For this project, nodes compromised the binary search trees, so a nested Node class was necessary with four attributes. Self.value holds a node's value and self.left and self.right served to reference the left and right children of the nodes. Unlike previous projects, the height was stored in a node attribute, self.height. Rather than keeping one updated height of the tree, each node held a number reflecting that node's height. Further information about the height will be explained in later methods. Runtimes of initialization of attributes are always constant, as a value is simply being assigned.

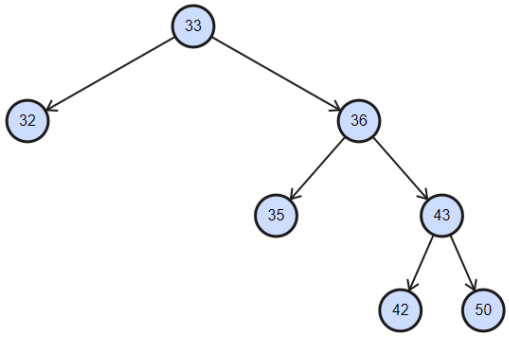
Initialization of the Binary Search Tree

One of the identifying characteristics of a binary search tree is the root attribute. In order to initialize a BST, a root attribute must be initialized and allocated as None. It is important to note that even when the BST has no nodes, it is still a binary search tree with a height of 0. The root attribute will be later assigned to a node. The runtime of this initialization is constant since the root attribute is being assigned a value.

Inserting an Element

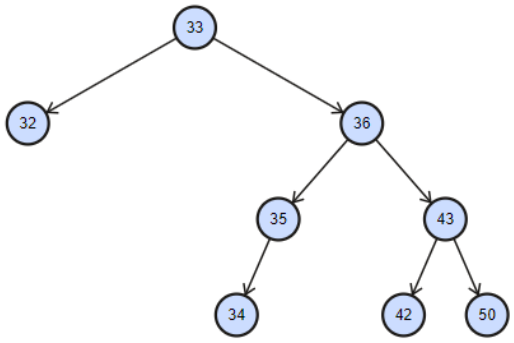
An important feature of this project requires allowing the user to modify the BST. Our first function to allow modification will be the insert_element method. With this public function, the user will be able to insert a value of their choice into the tree. It is important to note that this function will call a second private function which will perform the recursive steps, __rins. This private function demands two parameters, a specified node, t, and the value which the user wants to input, x. In line 16, the public function selects these two parameters to be the value the user inputs and the node with the root attribute. Python will then run the __rins method. This method will start with the root node and recur until it reaches the place of insertion, or the base case. Let's assume there are no nodes inserted into our tree yet. When our user tries to insert value 7 and the __rins function is called, Python will search for a node with the self.__root attribute, as demanded in the parameters. It will then realize that self.__root=None, and thus the first conditional in line 25 is met. This is our base case; it is the simplest step that we know how to do in any recursion. Line 26 will run, thus creating a Node t with the value 7 and a height attribute of one. The __rins method will then return t to the recursive call in the public function, and self.__root will be assigned to the node t.

Although inserting one node into the tree is quite simple, inserting nodes beyond the first requires a little bit more computational power. Let us assume we are inserting a node beyond the first such as in the tree below. Assume that we want to insert a value of 34. Conceptually, we know that we must travel to the right of 33 since 34>33. Then we must travel to the left of 36 since 34<36. Lastly, we know that we must travel to the left of 35 since 34<35. Since 35 has no children, we can create a node.



How does this transfer to code? When the user inputs the value as 34 and the private function is initialized, t will be the node with the root attribute. Because t is not None, it will skip over the first conditional onto the next. The next conditional demands that the user's value must be less than the value of the root node, so it will continue on to the next conditional. This third conditional (line 33) states that if the user's input value x is greater than the value of the root node, it must run the below functions. The below function is another recursive call. Thus, t will recur to the right node. Since recursions are essentially functions within functions, we now must go through the entire __rins method once more with the next function's t value, which will be the root node's right child (t.value=36). If we go through this function again, we will once again recur to the left. It is important to note that once the function reaches t.value=35, it recurs again to the left. However, the node with value 35 does not have any children, so its left attribute will be None. This will please the first conditional (our base case) as it did in the previous example when inserting the first node into the tree. Now that the base case function is "solved", the rest of our functions can also be "solved". In other words, each function will be able to return a value t until we once again are standing at our root node. Therefore, when the base case returns a t node with value 34 to the previous function, then a t node with value 35 will return to its previous function, and so on. Once the root node is returned to the public function, the self.__root attribute is reattributed to the node, and the entire tree will have formed it's links across its .left and .right attributes. Finally, the public function will always return the root.

There is one exception case in the private insertion method. In the case where the user inputs a number that is already in the tree, a Value Error is raised. One may wonder about the placement of the exception. When looking for a repeated value, we could walk throughout the entire tree or run through a list before calling the private recursion method. However, this would be quite inefficient in runtime and impractical. Rather, we can assume that the value isn't in the tree and run through our recursions. If the value being added to the tree is already in the tree, then we will come across it in our recursions. Before discovering this, I had created a conditional in the public function which would check for the value in a list with the tree's value. However, this caused almost every test function to crash, as I could not find a way to reset the list every time a new test was run. Although I had considered placing the Value Error conditional in the recursion method before attempting the list, my conditionals in the recursion method all began with if at the time. I quickly realized that the __rins method needed to have mutually exclusive conditionals because the base case would indirectly set t.value=x, thus triggering the ValueError. Changes were made in order to avoid such errors.



The runtime for insertion requires one to look at several functions. For example, the public insert_element function is calling upon the __rins function, and the __rins function calls the height_function in its worst cases. Thus, we must analyze all the functions each to understand the runtime of insertion. In its worst case, the __rins function will have to call the height_function. This function is constant, as Python is simply grabbing a known value, t.right.height or/and t.left.height, applying arithmetic to it, and reassigning t.height to that number. This will not affect the runtime of __rins because it is constant. The __rins function in its worse case will have linear time. If we have a completely unbalanced tree such as a linked list, we will have to recur through every single node to insert a leaf. Because the runtime will thus depend on how many nodes are already inserted, it will have a linear runtime. The public insertion function essentially does not do anything other than reassign the root attribute to the returned value of the private recursion function. Thus, it does not add to runtime. Overall, the insert_element function will thus have linear runtime in its worst case.

The Height Function in the Insert Function

In order to explain the rest of the __rins method, we must jump to the get_height and height_function methods. Finding the height of the tree seemed very challenging at the start; unlike heaps, the height does not rely on a mathematical method. Inserting and removing nodes does not guarantee that the height will change either. Thus, we had to find a way to count the nodes while removing or inserting them. Our game plan is as follows: each node will contain a height attribute which will be updated as nodes are inserted or removed. Let us go back to our last example of insertion. We inserted the value of 34 and labeled its height attribute as 1. What about the height attributes of the node above that? When the 34 node returns to the previous function, the previous node needs to change its height attribute too. Therefore, it calls the height_function for that node t. Assuming we are counting the height from the leaf nodes upwards, we need to guarantee that we are taking every possible pathway's height into account. Thus, the 35 node needs to look at both of its children's height attributes and decide which pathway is deeper. In other words, it needs to find the maximum height of its children to change its own height. The 35 node will look to its left and see a height of one, but when it looks to its right, it cannot find a height because t.right==None. Thus, it needs to take on the height of the left child+1. This is seen in the second conditional of the height_function method (line 167). When that recursive cycle returns t to the previous recursive cycle, node 36 will now have to reestablish a height value for itself. Since it has

both a left and a right child, it will need to compare its two children's height attributes and select the maximum. In this case, the first conditional will run (line 164), thus needing to find the maximum of 2 and 2. Luckily for us, the max function in Python will realize that these are the same number and will choose 2. 1 is added to the found maximum thus giving the 36 node a height of 3. When this recursive cycle returns to the previous function(which will now be our first recursive call), we will now have to update the root node's height attribute. Once again, we have two children, so it will look at its left child's height attribute (in this case it is 1) and at its right child's height attribute (in this case it is 3). The maximum between 3 and 1 is 3, so 1 is added to 3 and the new height is assigned to the root node. Although this has established the height of the tree, the height_function does not return the height to the user.

In order to get the height of the function, the user must call the get_height method. This method is compromised of two conditionals. The first conditional runs in constant time; if the root attribute is not assigned to a node, it returns a height of zero. In other words, if there are no initialized nodes, the height must be zero. The second conditional will also run in constant time. Because we have consistently updated the height attributes of each node in the tree, all we need to do is call the root's height attribute. This will be the overall height of the tree, as it is the top most node.

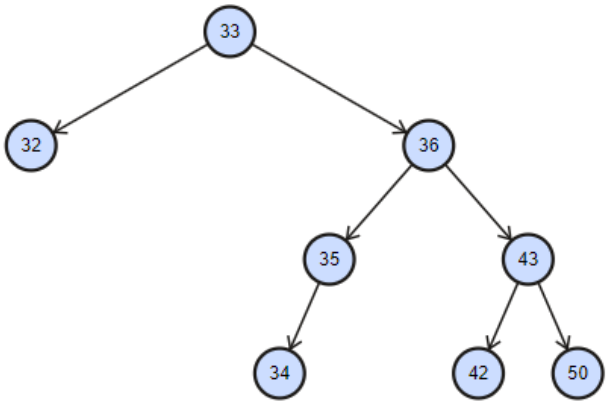
Although we have explained the height functions in context of the insertion functions, it should be recognized that this height algorithm will work in the exact same way for removal functions. Let us look at the removal algorithm.

Removing Elements

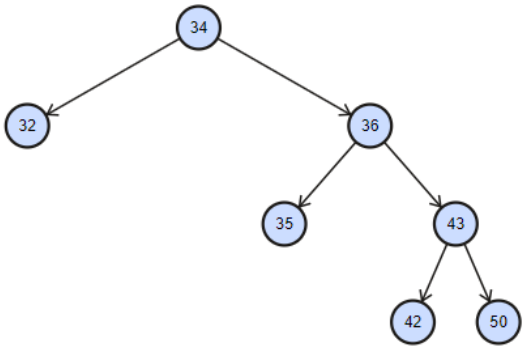
The removal methods are very similar to the insertion elements with the exception that we have multiple cases in our base case. The public removal function, remove_element, prompts the user to input a desired value to get rid of. The public function then calls the private function, __rrem, which once again takes parameters x and t. Like in the input functions, t once again is first set to self.__root. We have one exception in our recursive removal method; if the value input by the user is not found anywhere in the tree, we raise a Value Error. Otherwise, it will continue. Our first conditional describes our base case. Unlike the input function, our base case occurs when we have found the desired removal node, or when t.value==x. From here, we have several options. Let us assume that we are removing one node, 7, in a one node tree. Because the value of the node with the attribute self.__root matches the user input of 7, it will look at the nested conditionals. The first conditional is for nodes being removed that do not have children. Because we only have one node in our tree, this would be the appropriate conditional. Thus, we return the value None. When we return a function, the function ends. Thus, it reassigns self.__root to None in the public function.

What about our other cases? Let's assume we wanted to remove the root node when it had one left child. The first conditional in our base case would not work, but the second one would. Inside that conditional lies another set of nested conditionals. Because we have already told the interpreter that we have one node, all we need to do is tell Python which one it is. Because the root node's left child is not None, the first conditional is true. Thus, the left child is returned to the public function and set as the new root. This logic also follows for the right child. What happens if our root node has two children? Our base case would then fall under the third conditional, where neither left nor right children are None. Thus, we need to find the minimum value to the right of the root node. In this case, we only have one

child on the right. Let us reimagine the example in order to best understand the algorithm. Assume the tree looks like the one below and we are still removing the root node.



If we want to remove 33, we must look to its right child. This right child is set to a temporary variable called current. We must find the minimum value. We know that minimum values lie on the left side of nodes. Therefore, we must walk through as many left nodes as we can to reach the minimum. In lines 70-72, we perform a walk through the left-most nodes. The temporary variable is changed every time a smaller left child is found. This minimum value, 34, will take the place of the current t value, which in this case is the root value 33. However, we must still make sure to remove the extra 34 leaf node. Thus, the recursive removal function is called here, allowing that node to be removed. Although we have assumed that the node we are removing is the root node in order to explain the base cases, recursion will be necessary for cases in which the base case is not the root node. If we were trying to remove 50, we would need to travel down the tree in the same recursive manner that we did for insertion. The only difference between insertion and removal is the base case. The height and the recursive logic both remain the same.



The runtime of removals is very similar to the runtime of insertions. Because it depends on the public, private, and height_function methods, we must look at all of them. We previously mentioned that the height_function runs in constant time. We also mentioned that the worst case for the private insertion function was linear, due to its worst case being a linked list with the insertion being at the leaf node. This is also the same for the removal private function. In its worst case, we would have to remove the leaf node of the completely unbalanced BST. The extra conditionals in the base case do not add to the runtime. The public function will have linear runtime, as it depends on the private function and does not have any element that adds to runtime.

Tree Traversals

As we learned in class, there are three ways in which we can order a tree in an array or string. Because this project does not have a visual component, the tree traversals will serve to print the tree itself. We have three tree traversals: in order, pre order, and post order. All three algorithms are very similar in their code, as they are performing very similar things in different orders. In order to understand tree traversals, one needs to look at a tree as many small subtrees making up a big tree. Please note that this should not be taken literally, but it will serve to understand how recursions tie into this concept. Let us begin with in order traversals. The user will call a public in_order traversal function. This function will return an empty tree [ ] if there are no nodes. If there are initialized nodes, the function will create an empty list, call the private recursive in order function, and make changes to the formatting and type object. Once the public function calls the private recursive in order function, the recursive process begins. In in order traversals, we begin at the root node. We recursively go through the left-most nodes until t returns None. Once t is None, we are at the smallest possible function for which we have an answer to. This will return the function, thus throwing us back to the previous recursive call. We can then append this t value to the empty list. Then, another recursive function is called, and thus we must recur through the current t's right children. This continues until t is None, and thus the function is returned. With this combination of recursions, we can traverse throughout the tree and return the list to the public function. The public function then converts the list into a string and makes the adequate stylistic changes. Pre order traversals and post order traversals are almost identical in nature; the only difference is the order in which things are appended and recurred. In pre order traversals, we begin by appending the t value for each recursive call. This is why pre order traversals always begin with the root. Then, they recur left until they hit the base case, and lastly recur through the right children until they hit the base case. Post order will recur through al the left children until t is None, then through the right children until t is None, and will append the t value for each recursive call. It should be noted that there are as many recursive calls as there are nodes in the trees. In other words, the recursive methods will make sure every single node is appended to the list at their corresponding point. None will be left out.

The runtime for tree traversals depends on both the private and public functions. The private function has two recursive calls, thus leading the worst case runtime to be O(n). The public function simply makes changes to the string or list, which all run in constant time. Thus, the public function will be linear time since it depends on the private function. Because all three tree traversals have essentially similar logic, all will be linear time.

The String Method

The string method is used to print out the BST in the preferred tree traversal. Although the user can call any of the tree traversals, this method will avoid the program from crashing if the BST is printed and will make it easier to print out a tree. The string method will run in linear time since it depends on the tree traversal functions but does not add to runtime in any other manner.

BST_Test.py

The BST_Test file runs unit tests for several scenarios. There is a total of 71 test cases. The testing file is organized by traversal type, creating a total of three main sections. A subsequent height function section follows the three main sections, as height does not need a trasversal output. Each traversal section should have identical tests; the only difference in test is the ouput order. The two traversal subsections serve to test the overall functioning of insertion and removal.

Insertion Test Cases

The insertion test cases test the function of conditionals, recursion, and ValueErrors. My test cases make sure that the tree is empty before any nodes are inserted into it. I tested inserting at the root node, inserting at the left child, inserting at the right child, and creating multiple leveled, unbalanced trees. Although this was not included in the unit test, all parts of the insertion functions were shown to run by inserting print statements. In this manner, I could not only conceptualize how the algorithm was working, but actually see it. Repeated Value Insertions were tested in order to induce a ValueError. Both floats and strings were tested in the cases in order to test other object types and the functioning of mathematical signs such as <,>, and =.

Removal Test Cases

Due to the many conditionals of the removal test cases, this subsection was much larger. Value Errors were tested by removing elements when there were no nodes in the tree, by removing elements that had already been removed in the tree, and when the value did not exist in the tree. Normal functioning of the remove function was also tested, such as removing the root of a one node tree, removing a leaf node, removing a node with a left child, removing a node with a right child, and removing a node with two children. All base cases seemed to function properly. Other cases tested include removing every single node and creating combinations of insertion and removal of nodes. This was done to test recursion and to make sure each function worked when repeated and when placed in any order. Removals were also tested with print functions outside of the unit testing to ensure proper functioning. Some Removal and Insertion functions include height checks.

Height Functions

The height functions essentially repeat several of the test cases in insertion and removal. They test to make sure that the height only changes under specific conditions rather than every time a node is inserted and removed.

Works Cited

All images were created using this BST generator: http://btv.melezinek.cz/binary-search-tree.html