CSCI 241 Data Structures
Project 4: It's Just a Jump to the Left and a Step to the Right

In this project, you will implement a basic unbalanced binary search tree and then add balance features to obtain optimal height.

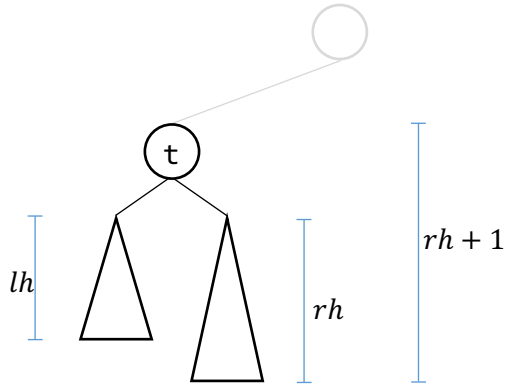
Part I – Unbalanced Tree

Your primary focus in this part is recursion. Conceptually, binary search trees are fairly simple structures, so let's take advantage of this implementation to develop our recursion skills. Review the recursive algorithm description in the BST slide deck very carefully. The wording is deliberate, and your implementation must follow the presented algorithms. We also suggest reviewing the associated videos to ensure that you understand the concepts before beginning your implementation.

When considering the traversal methods (which must also be recursive), remember that recursion works by dividing the problem into smaller components, then combining the smaller results to form the larger solution during the return path. Strings can be concatenated together to form larger strings using the + operator. Review the traversal algorithms and consider how the process of building the string can be divided into smaller steps and how those smaller strings can be combined to form the larger result. As one example, notice that the in-order traversal of a subtree rooted at node `t` is the concatenation of three strings: the in-order traversal of the subtree rooted at `t`'s left child, `t`'s value, and the in-order traversal of the subtree rooted at `t`'s right child.

Your implementation will also provide a `get_height()` method to obtain the number of levels in the tree. Note that this method is specified to operate in constant time. This means that height cannot be computed on demand, as counting the levels yields linear-time performance. Instead, add an attribute to the `__Node` class to store the height of the subtree rooted at that node. Just before returning a node reference at the end of a recursive call, update that node's height field to be correct. If you do this before each return, then you know at all times that the height field of every node below you in the tree has the correct value. Because the heights of the subtrees rooted at `t`'s children are now known to be correct, we can say that `t`'s height is equal to the maximum of its children's heights (accessible in constant time through child node attributes) plus 1. An important consideration here is that a non-existent subtree has height 0 (be careful not to crash in this case). Also notice that the height of the subtree rooted at a newly created node object is always 1.

In the example below, `t`'s height should be updated to $rh + 1$ before it is returned. We choose rh because the right subtree's height is larger than the left subtree's height. If every node in the tree stores the height of the subtree rooted at that node, then you can return the height of the entire tree in constant time.



For one final reminder, notice that you cannot make the skeleton methods that we provide recursive. Because the only parameter that they take is the value being added or removed, and you cannot change the signatures of the public methods, you will have to introduce private accessory methods that are recursive. We did this in class. When asked to insert 50 into a BST, our public `insert(50)` method calls `__recursive_insert(50, self.__root)`. That second parameter (`t` in my slides) is the recursion control variable that approaches the base case (`t` is the node representing the root of a smaller and smaller subtree until we hit a base case).

Additional details specific to each method including required exceptions are presented in the skeleton file.

Milestone Checkpoint: submit your unbalanced BST to gradescope for testing (this does not affect your grade, but it ensures that your tree construction is correct before you attempt balance operations.

Part II – AVL Balancing

In this part, you will extend your BST implementation in Binary_Search_Tree.py to guarantee $O(\log n)$ insertions and removals. The changes required to support this are minimal. Provide a private method called `__balance(t)` that takes a node `t` as a parameter and treats it as the root of subtree. If the subtree rooted at `t` is unbalanced, rotate as necessary to balance it and return the new root of the now balanced subtree. Because this balance operation will be invoked on the return path from recursive insertion and removal, you can be sure that everything below `t` is already balanced. This means that checking to see if the tree rooted at `t` is balanced (and rotating it if it is not) is a constant time operation. You may also wish to introduce other additional private methods, as well.

If you followed the algorithms presented on the lecture slides and in class, all you have to do after writing the `__balance(t)` function is change the last line in your private recursive insert and remove methods from `return t` to `return self.__balance(t)` and reconsider when you compute height. Note that it is not necessary to check the balance of the new node created and returned in the base case, because it will always be balanced already. Be sure that when you return a

subtree it is balanced and the height attribute of every node at or below `t` is correct (but only reevaluate the heights of nodes whose subtrees could potentially have changed—that is, every node on the insertion/removal path and every node actively involved in a rotation).

Once you have balanced insertions and removals implemented, **add another public/private method pair for recursively constructing a Python list of the values** in the tree. This method should work just like the in-order traversal methods, but it should return a python list, not a string. Your public method should be called `to_list`; you are free to name your private recursive method whatever you like.

Finally, complete the implementation of the provided Fraction class by implementing the three comparison operators. The main section of this program should create a Python list of fraction objects, then insert them one at a time into an initially empty AVL tree, then get the in-order Python list representation using the new `to_list` method of Binary_Search_Tree, showing that the returned list is in sorted order.

Writeup Prompts

For each of the following questions, provide a prose response not more than one page in length. Each response should be on a separate page, and the top of each page should specify the question being answered.

1) What is the worst case performance of `insert_element`, `remove_element`, and `to_list` for the balanced BST? Provide an explanation to justify each performance class. Note that if a method calls other methods, you should include the entire operation in your performance analysis. For example, the runtime of insert_element must include the runtime of your private recursive insert function.

2) What steps have you taken to ensure that your methods work properly in all cases? Your discussion should include the `insert_element`, `remove_element`, and `to_list` operations for the BST as well your Fraction class methods.

3) What is the performance of the sorting method implemented in this project? Be certain to account for all steps. Does the sorting performance change depending on the types of objects we are sorting?

(submission expectations next page)

SUBMISSION EXPECTATIONS

`Binary_Search_Tree.py` This should be your implementation of an AVL tree. You are free to add additional private support methods (in fact, this is necessary), but do not change the public interface to this class other than introducing the new `to_list` method.

`BST_Test.py` Your unit tests for implementations. No skeleton file is provided for this component. For testing, notice that the three traversals (in-order, post-order, and pre-order) uniquely identify a binary search tree. No two unequal trees share all three traversal orderings. Ensure that your traversals work correctly and use the combination of all three of them to test the structure of the tree after insertion and removal operations.

`Fraction.py` The provided Fraction class with the comparison methods implemented and with the main section sorting a Python list of fraction objects.

`MethodPerformance.pdf` A prose writeup briefly presenting your response to writeup prompt 1 above.

`Testing.pdf` A prose writeup briefly presenting your response to writeup prompt 2 above.

`SortingPerformance.pdf` A prose writeup briefly presenting your response to writeup prompt 3 above.