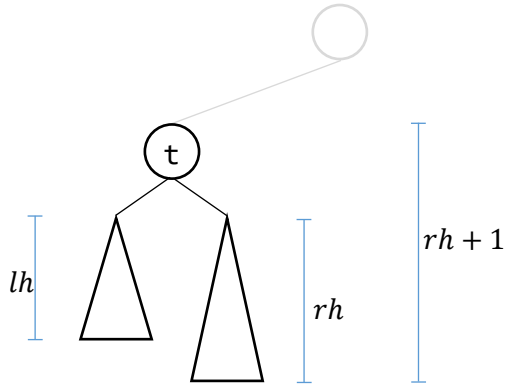
CSCI 241 Data Structures
Project 4: It's Just a Jump to the Left and a Step to the Right

In this project, you will implement a basic unbalanced binary search tree. Your primary focus in this project 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.

SUBMISSION EXPECTATIONS

`Binary_Search_Tree.py` This should be your implementation of a basic unbalanced binary search 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.

`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 them to test the insertion and removal methods.

`Writeup.pdf` A prose writeup explaining the purpose and efficacy of your test cases and an analysis of the worst-case asymptotic performance of every method in your implementation. Note that if a method calls other functions, 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.