Project 5: AVL Trees

Due: Thursday, February 27th @ 8:00pm

This is not a team project, do not copy someone else's work.

Note: This project has been proven to be the most difficult one for students, please start as soon as you can.

Description

In this project, you will be implementing an AVL Tree. An AVL Tree is a specific type of Binary Search Tree. A Binary Search Tree is a structure in which there are nodes, and each node has at most two children, a left and a right node. The rule for a Binary Search Tree is that the left node of every node in the tree must either be less than the parent node, or not exist, and the right node of every node in the tree must either be greater than the parent node or not exist. A tree structure is hierarchical, unlike many other structures that are linear such as linked lists, queues, and stacks. Tree structures are often used due to many operations requiring less time than linear structures. For example, std::map in C++ is actually implemented using a red-black tree. An AVL Tree is a self-balancing Binary Search Tree, which means that no subtree of the overall tree will be unbalanced (the difference of heights of subtrees can be at most 1). Your assignment is to complete the remaining functions of the AVL Tree structure and solve an application problem about trees.

Turning It In

Your completed project must be submitted as a folder named "Project5" compressed in a zip file that includes:

- AVLTree.py, a python 3.5 or higher file.
- __init__.py
- README.txt, a textfile that includes:
    - Your name
    - Feedback on the project
    - How long it took to complete
    - A list of any external resources that you used, especially websites (make sure to include the URLs) and which function you used this information for. Please note that you can not use outside resources to solve the entire project, or use

outside sources for multiple functions that lead to the solution of the project, then you are submitting someone else's work not yours. Outside sources use should not be used for more than one function, and it should not be more than a few lines of code to solve that function. Please note that there is no penalty for using the programming codes shared by Onsay in Lectures and posted on D2L as well as in Zybooks.

Assignment Specifications

class TreeNode: **(Do not modify this class)**

- **value**-> the value stored in this node [required for instantiation]
- **parent**-> the parent of this node (defaults to None)
- **left**-> the left child of this node (defaults to None)
- **right**-> the right child of this node (defaults to None)
- **height**-> the height of the tree rooted at this node. (always zero upon creation)
- **__init__()**
- **__eq__()**
- **__str__()**

class AVLTree:

- **root**-> the value stored in this node (always None upon creation)
- **size**-> number of TreeNodes in the AVLTree (always zero upon creation)
- **__init__()**
  - Do not modify
- **__eq__()**
  - Do not modify
- **__str__()**
  - Returns a string that represents a visual of the AVL tree and is provided for your own personal use
  - Below is an example of its implementation

- **insert**(self, node, value):
  - Takes in value to be added in the form of a node to the tree
  - Takes in the root of the (sub)tree the node will be added into
  - Do nothing if the value is already in the tree
  - Balances the tree if it needs it
  - **Must be recursive**
  - O(log(n)) time complexity, O(1)* space complexity
- **remove**(self, node, value):
  - Takes in value to remove from the tree
  - Takes in the root of the (sub)tree the node will be removed from
  - Do nothing if the value is not found

    - When removing a value with two children, replace with the maximum of the left subtree(hint: implement max/min before this function)
    - Return the root of the subtree
    - Balances the tree if it needs it
    - **Must be recursive**
    - O(log(n)) time complexity, O(1)* space complexity
- **height**(node): **--> static method**
    - Returns the height of the given node
    - O(1) time complexity, O(1) space complexity
- **update_height**(node): **--> static method**
    - Updates the height of the node based on its children's heights
    - O(1) time complexity, O(1) space complexity
- **depth**(self, value):
    - Returns the depth of the node with the given value, otherwise -1
    - O(height) time complexity, O(1) space complexity
- **search**(self, node, value):
    - Takes in value to search for and a node which is the root of a given (sub)tree
    - Returns the node with the given value if found, else returns the potential parent node
    - **Must be recursive**
    - O(log(n)) time complexity, O(1)* space complexity
- **inorder**(self, node):
    - Returns a **generator object** of the tree traversed using the inorder method of traversal starting at the given node
    - Points will be deducted if the return of this function is not a generator object (hint: **yield** and **yield from**)
    - **Must be recursive**
    - O(n) time complexity, O(n) space complexity
- **preorder**(self, node):
    - Same as inorder, using the preorder method of traversal
    - **Must be recursive**
    - O(n) time complexity, O(n) space complexity
- **postorder**(self, node):
    - Same as inorder, using the postorder method of traversal
    - **Must be recursive**
    - O(n) time complexity, O(n) space complexity
- **bfs**(self):
    - Same as inorder, using the breadth-first method of traversal
    - Allowed to import queue to complete this method only
    - O(n) time complexity, O(n) space complexity
- **min**(self, node):
    - Returns the minimum of the tree rooted at the given node
    - **Must be recursive**
    - O(log(n)) time complexity, O(1)* space complexity
- **max**(self, node):
    - Returns the maximum of the tree rooted at the given node

    - **Must be recursive**
    - O(log(n)) time complexity, O(1)* space complexity
- **get_size**(self):
    - Returns the number of nodes in the AVL Tree
    - O(1) time complexity, O(1) space complexity
- **get_balance**(node): **--> static method**
    - Returns the balance factor of the node passed in
    - Balance Factor = height of left subtree – the height of right subtree
    - O(1) time complexity, O(1) space complexity
- **set_child**(parent, child, is_left): **--> static method**
    - sets the parent node's child to the child node
    - is_left is a boolean that determines where the child is to be placed in relation to the parent where True is left and False is right
    - O(1) time complexity, O(1) space complexity
- **replace_child**(parent, current_child, new_child): **--> static method**
    - Makes the new_child node a child of the parent node in place of the current_child node
    - must use **set_child**
    - O(1) time complexity, O(1) space complexity
- **left_rotate**(self, node):
    - Performs an AVL left rotation on the subtree rooted at node
    - Returns the root of the new subtree
    - must use **replace_child** and **set_child**
    - O(1) time complexity, O(1) space complexity
- **right_rotate**(self, node):
    - Performs an AVL right rotation on the subtree rooted at node
    - Returns the root of the new subtree
    - must use **replace_child** and **set_child**
    - O(1) time complexity, O(1) space complexity
- **rebalance**(self, node):
    - Rebalances the subtree rooted at the node if needed
    - Returns the root of the new, balanced subtree
    - must use **left_rotate** and **right_rotate**
    - O(1) time complexity, O(1) space complexity

Application Problem

- **is_avl_tree**(node):
    - This function will initially take in the root of a binary search tree
    - Returns True if the given tree is a valid AVL tree, otherwise False
    - You are only allowed to use the TreeNode attribute's left and right, any additional use of methods or attributes will result in a zero.
    - You will not need to worry about verifying the nodes' values
    - **Must be recursive**
    - O(nlog(n)) time complexity, O(1)* space complexity

- *Note: the **is_avl_tree** function will not depend on any of the code that you were required to implement for the AVLTree class. The tree is made using only value, left and right. This means that the parent, height, and depth are not updated. If you are stuck on the AVLTree code, you can still pass this test case as it is independent. Refer to Mimir testing.*

Manual Complexity Grading

Functions	**Time**		**Space**	
insert/remove	O(log(n))	4	O(1)*	2
height/_update_height/ get_size/get_balance/set_child/ replace_child/rotates/rebalance	O(1)	4	O(1)	2
depth/search/min/max	O(log(n))	4	O(1)*	2
traversals	O(n)	4	O(n)	2
application	O(nlog(n))	4	O(1)*	2
Total		**20**		**10**

* Not taking into account space allocated on the call stack

Example of **__str__(self)** implementation:		

Code:
`a = AVLTree()`
`for i in range(10):`
`  a.insert(a.root, i + 1)`
`print(a)`
	When you print your tree, you will see a visual representation of it in your terminal.	Feel free to manipulate the TreeNode attributes in the code in parenthesis. It is set to parent and height right now. (you will have to alter __str__(self))

```
                Visual Level Order Traversal of AVL Tree - Node (Parent Height)

Level 0:                                        4 (None 3)
Level 1:                  2 (4 1)                                       8 (4 2)
Level 2:       1 (2 0)              3 (2 0)              6 (8 1)              9 (8 1)
Level 3:                                          5 (6 0)    7 (6 0)              10 (9 0)
```

Assignment Requirements

- You are required to complete the docstrings for each function that you complete.
- You are provided with skeleton code for the AVLTree class and you must complete each empty function.
- You may create additional helper functions and are required to complete docstrings for those as well.
- You can only assign a node's height within update_height. You may not manually update it anywhere else outside of that method.
- Do not modify function signatures in any way.
- **Make sure that you are adhering to all specifications for the functions, including time and space complexity as well as whether or not the function is recursive.**

Additional Notes

- Recommendation: start on the methods related to rebalancing and rotation.
- Many methods depend on insertto work. For instance, even with the correct searchalgorithm, the test case may not pass unless insertion is correct.
- The BFS method does not have a node parameter as the other three depth-first traversals do because of the fact that it is iterative. Because the DFS traversal

methods are recursive, we will need to pass in a node parameter in order to keep track of where we are while traversing.

- Remember that with a given list, pop(0) is an O(n) operation.
- While the functionality of set_child and replace_child are required, the functions themself are not commonly used components to the left/right rotates. Rather, they avoid the reuse of code and help compartmentalize the rotate's algorithm which ultimately aids in understanding.

Helpful Links

- Visualizing AVL Trees
  - https://visualgo.net/bn/bst (remember to click on AVL)
  - https://www.cs.usfca.edu/~galles/visualization/AVLtree.html
- Static Methods
  - https://pythonbasics.org/static-method
- Yield
  - https://docs.python.org/2.4/ref/yield.html
  - Example:
    - Both yield a generator object from 0-9

```
def generator():
    for i in range(10):
        yield i

def generator2():
    yield from generator()
```

Rubric	Points
Visible + Hidden Test Cases	56
Fully Hidden Test Cases	12
Manual Grading	30
README.txt	2

Total Points	100

Project written by Anna De Biasi