Project 5: AVL Trees

Due: Thursday, March 14 @ 8:00pm
This is not a team project, do not copy someone else's work.

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. An AVL Tree is a Binary Search Tree that self-balances, 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" and must include:

- AVLTree.py, a python 3.5 or higher file.
- 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 leads 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 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.

Assignment Specifications
The Node class is fully implemented and provided for you. **Do not modify this class.** We have provided the __init__() and __eq__() methods in the AVLTree class, **do not modify these methods.** Your task will be to complete the methods listed below in the AVLTree class that have not been completed for you. Make sure that you are adhering to the time and space complexity requirements. **Do not modify function signatures in any way.**

- **insert**(self, node, value):
  - Takes in a 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
- **Must be recursive**
- Balances the tree if it needs it
- O(log(n)) time complexity, O(1)* space complexity

- **remove**(self, node, value):
- Takes in a 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
- Return the root of the subtree
- **Must be recursive**
- Balances the tree if it needs it
- O(log(n)) time complexity, O(1)* space complexity

- **search**(self, node, value):
- Takes in a value to search for and a node which is the root of a given tree or subtree
- Returns the node with the given value if found, else returns the potential parent node
- 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 **from**)
- **Must be recursive**
- O(n) time complexity, O(n) space complexity

- **preorder**(self, node):
- Same as inorder, only using the preorder method of traversal
- O(n) time complexity, O(n) space complexity

- **postorder**(self, node):
- Same as inorder, only using the postorder method of traversal
- O(n) time complexity, O(n) space complexity

- **depth**(self, value):
- Returns the depth of the node with the given value
- O(height) time complexity, O(1) space complexity

- **height**(self, node):
- Returns the height of the tree rooted at the given node
- O(1) time complexity, O(1) 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**(self, node):
  - Returns the balance factor of the node passed in
  - Balance Factor = height of left subtree – height of right subtree
  - O(1) time complexity, O(1) space complexity
- **left_rotate**(self, root):
  - Performs an AVL left rotation on the subtree rooted at root
  - Returns the root of the new subtree
  - O(1) time complexity, O(1) space complexity
- **right_rotate**(self, root):
  - Performs an AVL right rotation on the subtree rooted at root
  - Returns the root of the new subtree
  - O(1) time complexity, O(1) space complexity
- **rebalance**(self, node):
  - Rebalances the subtree rooted at node, if needed
  - Returns the root of the new, balanced subtree
  - O(1) time complexity, O(1) space complexity

In addition to the functions inside the AVL Tree class, you will be required to complete the following function:

- **def repair_tree**(tree):
  - Takes in a tree where two values may have been swapped, violating the BST property of nodes on the left being less than the parent node, and nodes on the right being larger than the parent node
  - Repairs the tree by finding if two values have actually been swapped, and swapping them back if necessary
  - O(n) time complexity, O(n) space complexity

    * Not taking into account space allocated on the call stack

Assignment Notes

- 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 use more functions if you'd like, but you must complete the ones given to you. If you do choose to make more functions, you are required to complete docstrings for those as well.
- The function AVLTree.visual() is provided for you to help with debugging, it displays the tree in forms of levels, with nodes represented as tuples in the form of (node.value, node.parent). Example:
  - Level 0: (10, None)
  - Level 1: (5, 10) (15, 10)
  - Play around with it to get a feel for what it represents.
- Make sure that you are adhering to **all** specifications for the functions, including **time and space complexity** and whether or not the function is recursive
- A great resource is https://www.cs.usfca.edu/~galles/visualization/AVLtree.html , this is an AVL tree visualizer that shows you what your tree should look like after insertions and deletions

- https://docs.python.org/2.4/ref/yield.html - Here is documentation on the yield keyword in python. Yield is like return, except it adds to the generator object. "Yield from" yields from the yield of whatever you're yielding from. That was kind of complicated. Here's an example:
  - def generator():
    - for i in range(10):
    - yield i //yields a generator object
  - def generator2():
    - yield from generator() //yields a generator object from 0-9, also!

Rubric

Mimir Testcases: __ / 70

Time Complexity Requirements:

- insert/remove __ / 16
  - O(log(n))
- height/rotates/rebalance/get_balance __ / 4
  - O(1)
- search/depth/min/max __ / 4
  - O(log(n))
- traversals __ / 6
  - O(n)

Project authored by Brandon Field