Due: Nov. 16, 2022 at 11:59pm

Overview

This lab focuses on implementing a Binary Search Tree.

Data Structures

Binary Search Trees are node-based binary tree data structures satisfying the following criteria:

- The left subtree of a node contains only nodes with keys that are less than the current node's key.
- The right subtree of a node contains only nodes with keys that are greater than the current node's key.
- The left and right subtrees of a node must each be a binary search tree.

You should fill out all of the methods in the provided skeleton code **lab3.py**. You may add additional methods but should not add any private variables to the class. Like before, you should not alter any names for any of the classes, methods, or files. Your code will be graded using an autograder on Gradescope. To start you off, some test cases have been provided in **test˙lab3.py**. You may not use any data structures from the Python standard library. Some inbuilt functions in python can be used as required.

Tree

`insert(self, data)` : All insertions will be done with integer data. `insert()` should navigate the tree while comparing `data` with each node encountered to determine if your traversal should take a left or a right. When you arrive at an empty leaf you should replace the NoneType placeholder with a **Node** containing the inserted value.

`__traverse(self, curr_node, traversal_type)` : A helper function for implementing node traversals. Traversal will be implemented using generators[1]. You may use additional helper methods, but all three traversals can be achieved using the single `__traverse()` method. `traversal_type` dictates the order of the traversal according to the following:

- in-order: (Left, Root, Right),
- pre-order: (Root, Left, Right),
- post-order: (Left, Right, Root).

[1]Generators are a type of iterable (like a list or a dictionary in Python) which you can only iterate over one time. Unlike lists or dicts, generators do not store their values in memory but instead generate values when told to.

`__find_node(self, data)` : Given an `int data`, `__find_node()` returns the node in the tree which holds `data`. If no such node exists in the tree, return **None**. The double underscores that precede the method name indicate this method is private and can only be used from within in **Tree** class.

`contains(self, data)` : Similar to `__find_node()`, this method is public. Rather than returning the node corresponding to `data`, `contains()` instead returns `True` if some node in the tree contains `data` and it returns `False` otherwise.

`find_successor(self, data)` : Find and return the node containing the next highest value from `data`. This method is useful in implementing the `delete` method but may also be called on its own. From the structure of the binary search tree we know the following:

- If the right subtree of the node is nonempty, then the successor is just the leftmost node in the right subtree.
- If the right subtree of the node is empty, then go up the tree until a node that is the left child of its own parent is encountered. The parent of the encountered node will be the successor to the initial node.

  Note: `Raise KeyError` if finding a successor of a node that does NOT exist in the tree

`delete(self, data)` : delete the node associated with `data` from the tree.

- If said node has no children, simply remove it by setting its parent to point to `None` instead of it.
- If said node has one child, delete it by making the node's parent point to the node's child.
- If said node has two children, then replace it with its successor, and remove the successor from its previous location in the tree.
- If the value specified by `delete()` does not exist in the tree, then the structure is left unchanged.

Note: `delete()` returns `None`.
Note: `Raise KeyError` if deleting a node that does NOT exist in the tree.

Note**: `delete()` is a fairly complex method to implement so rigorous testing will be necessary for getting it right. I give this warning so that you don't leave `delete()` for last thinking it will be simple and doable the day before the lab is due.

`min(self)` : Returns the value of the node with the minimum value in the tree. Return `None` if the tree is empty.

`max(self)` : Returns the value of the node with the maximum value in the tree. Return `None` if the tree is empty.

Testing

Some sample test cases have been provided in **test˙lab3.py** but this provided list is not exhaustive. You should provide one test case for each of the following methods:

`insert()`

`__traverse()`

`__find_node()`

`contains()`

`find_successor()`

`delete()`

Additionally, provide two more tests to check that calling `delete()` or `find_successor()` on an empty tree results in `KeyError` being raised. It is strongly encouraged that you write more of your own tests.

Submission

Compress the **lab3.py** and **test˙lab3.py** files and upload in Gradescope similar to the previous programming assignments. The test cases file should contain all the additional test cases that you have written to test your code.

Grading

The assignment will be graded as follows:

— AutoGrader - [70pts]

— Style - [10pts]

— Additional Test Cases - [15pts]

-2pt- **Test Case:** Create test for `insert()` method.
-2pt- **Test Case:** Create test for `__traverse()` method.
-2pt- **Test Case:** Create test for `__find_node()` method.
-2pt- **Test Case:** Create test for `contains()` method.
-2pt- **Test Case:** Create test for `find_successor()` method.
-2pt- **Test Case:** Create test for `delete()` method.
-3pt- **Test Case:** `KeyError` is raised when `find_successor()` or `delete()` are called on an empty tree.

— DocStrings - [5pts]

-5pt- **DocString:** Provide documentation for the **tree** class in **lab3.py**. This means the docstring should contain a description of the class along with its methods, attributes, and any errors that is raises.