Mod 9 Lab: Binary search trees and generators

Description

We will add some functionality to a binary search tree (BST) set in this lab, including using a generator for traversal. Our typical BSTs store key:value pairs. A BST set will just store keys.

The textbook includes the full code for a binary search tree, including generator-based traversal. Avoid looking at the textbook during lab, but feel free to reference it afterwards if you are stuck. Spend the next 75 minutes trying to solve this problem with your lab partner.

Tree Traversal

Iterators

Tree traversal is inherently recursive - we need to go left until we hit `None`, then right until we hit `None`, with a base case somewhere in the mix. Importantly, we need to keep track of the recursive stack of parent nodes.

Classical iterators are not a great tool for recursive tree iteration, because any local variables (like a recursive stack) are lost as soon as an object is returned. The easiest method is to take a long time on our first iteration, recursively going through the entire tree to build a traversal queue, before we start returning items:

```
# assume we want to iterate using in-order traversal
class BSTNode_Iterator:
    def __init__(self, node):
        self.queue = []
        self.in_order(node) # build up the entire queue during initialization
        self.counter = 0

    # recursive in-order traversal to build up the queue
    def in_order(self, node):
        if node.left is not None: self.in_order(node.left)
        self.queue.append(node)
        if node.right is not None: self.in_order(node.right)

    # once the queue is finished, we can iterate through it
    def __next__(self):
        if self.counter < len(self.queue):
            self.counter += 1
            return self.queue[self.counter-1].key

        raise StopIteration
```

The queue initialization can be very slow for large trees. Ideally, we would like to "return" as soon as we find the next appropriate node, while keeping track of our current recursive stack.

Generators

Python uses a special keyword `yield` to return a value, while keeping track of the current state (e.g. all local variables) for the next call to a function. Functions that use `yield` instead of `return` are called generator functions.

```
>>> def gen_func(x, _max):
>>>     while x < _max:
>>>         yield x    # return this value, but remember the value of x
>>>         x += 1
>>>
>>> for item in gen_func(5, 8):
>>>     print(item)
5
6
7
>>>
```

Generator functions:

- are defined by using the keyword `yield`
- remember local variables between calls
- begin where they left off on subsequent calls (from the line after the most recent `yield`)
- return **generators**, a special type of iterator. This means generator functions can be used in for loops

There is a lot going on here conceptually, and it may take a few read throughs and some experimentation to fully internalize all of it. The conceptual load is worth it though - generators make it almost trivial to traverse trees, because we can keep track of a recursive stack between `yield` statements.

Deliverables

Most of your work will be adding methods to the class `BSTNode`.

- `put`
  - adds a specified key to the BST
  - ignores duplicate keys
- `post_order`
  - iterates through the subtree rooted at this node in post-order
- `pre_order`
  - iterates through the subtree rooted at this node in pre-order

You will also need to make minor modifications to the public-facing class `BSTSet`, so the examples given below appropriately call the `BSTNode` methods.

Do not add any rotation/weight balancing to your BST for this assignment; that will cause the pre-order and post-order autograder tests to fail.

Examples

Any examples below are intended to be illustrative, not exhaustive. Your code may have bugs even if it behaves as below. Write your own tests, and think carefully about edge cases.

Some other test cases that may be helpful (not for a grade, but they'll help you debug):

- a balanced tree with the integers 0-7. What order should you add keys to get this result?
- trees where you have added the same key multiple times
- trees with non-standard keys (negative integers? characters?)

```
# unbalanced tree
# 0
#  `-1
#     `-2
#        `-3
bst = BSTSet()
for i in range(4): bst.put(i)

# test in-order
ino = []
for key in bst.in_order(): ino.append(key)
assert ino == [0, 1, 2, 3]

# test pre-order
preo = []
for key in bst.pre_order(): preo.append(key)
assert preo == [0, 1, 2, 3]

# test post-order
posto = []
for key in bst.post_order(): posto.append(key)
assert posto == [3, 2, 1, 0]
```

Submitting

At a minimum, submit `BSTSet.py` and `BSTNode.py`.

Students must submit **individually** by the due date (typically, Friday at 11:59 pm EST) to receive credit.