Lab 5

In this lab, you will implement a binary search tree and explore the use of `yield` to iterate over trees.

1 Binary Search Tree

1.1 Definition

A Binary Search Tree is a binary tree where, for each node in the tree:

- the value of every node to the left *must* be smaller, and
- the value of every node to the right *must* be larger.

For our implementation in this lab, we will allow multiple of the same value to be inserted into the tree, so we will modify the above definition slightly where for each node in the tree:

- the value of every node to the left *must* be smaller, and
- the value of every node to the right *must not* be smaller (i.e., is larger or equal).

1.2 Operations

In the file `bst.py`, you will implement the following functions operating on binary search trees:

- `is_empty` This function takes a BST as an argument and returns whether or not the tree is empty.
- `search` This function takes a BST and a value as arguments and returns whether or not the value is in the tree.
- `insert` This function takes a BST and a value as arguments and inserts the value into the proper location in the tree. That is, it will insert into the left sub-tree if the value is smaller than the current node's value, and the right sub-tree otherwise (e.g., duplicates will be stored on the right).

  This function returns the resulting tree.
- `delete` This function takes a BST and a value as arguments and deletes the value from the tree (if present) while preserving the binary search tree property that, for a given node's value, the values in the left sub-tree must be smaller, and the values in the right sub-tree must not be smaller.

  This function returns the resulting tree.
- `find_min` This function takes a BST as an argument and returns the smallest value in the tree. If the tree is empty, this function should raise a `ValueError`.
- `find_max` This function takes a BST as an argument and returns the largest value in the tree. If the tree is empty, this function should raise a `ValueError`.
- `height` This function takes a BST as an argument and returns the height of the tree. The height of a tree is measured as the length of the longest path from the root to a leaf. For our purposes, we'll say that the height of an empty tree is -1[*].

[*]This is a slightly controversial decision, but it makes the code easier to write.

Note that you can (and should) assume that all items added to your list are comparable with the `<` operator and can be compared for equality with `==`. You cannot make *any* other assumptions about the items in your list.

1.3 Testing

In a file named `bst_tests.py`, write test cases for the functions above.

2 `yield`

2.1 The `yield` statement

Python supports a `yield` statement to, roughly, provide a "return" value by suspending the function. Execution of the suspended function will continue after the `yield` when another result is requested from the function. This is done, for instance, by iterating through the yielded values in a loop.

For example, consider the function `seven_nine` below. It will yield the 7 first and then yield the 9 if another result is requested. You should run this code in the interpreter and modify it to yield additional values to make sure the functionality is clear.

```python
def seven_nine():
    yield 7
    yield 9


for value in seven_nine():
    print(value)
```

2.2 Further Details

More accurately, in Python, the `yield` statement creates what is called a generator. Generators support an operation to retrieve the next generated value; it is this operation that triggers the initial execution of the function code and each subsequent execution following a `yield`. This can be seen more explicitly if you run the following code:

```python
def seven_nine():
    yield 7
    yield 9


my_gen = seven_nine()
print(type(my_gen))

print(next(my_gen))
print(next(my_gen))
print(next(my_gen))  # will raise a StopIteration exception
```

2.3 yield from

Python also allows you to `yield` all the values `from` a different iterator. This is possibly best seen through an example. Suppose that we want to `yield` all the values from 1 up to (and including) a given n. We could do this will a loop:

```
def sample_iterator(n: int):
    for val in range(1, n + 1):
        yield val


# This will print the numbers from 1 to 10.
for value in sample_iterator(10):
    print(value)
```

But, we can also think of this as yielding all the values from 1 up to 9, and *then* yielding 10. Which is to say, we could think of this recursively! Do we have a function to `yield` all the values from 1 up to 9? Yes! It's the one we're writing!

```
def sample_iterator(n: int):
    if n > 0:
        yield from sample_iterator(n - 1)
        yield n


# This will also print the numbers from 1 to 10.
for value in sample_iterator(10):
    print(value)
```

You'll get a chance to try this out for yourself in the next section.

3 Tree Traversals

It is often desirable to access every element of a tree for further processing. For instance, one might wish to print every element, to write every element to a file, to insert every element into a database, or to perform some computation on every element. A tree traversal strategy specifies the order in which the nodes of a tree are visited, where "visited" in this case amounts to processing of the data stored within the node.

Instead of writing a separate function for each traversal task (i.e., for printing, for inserting into a database, etc.), we will hide the traversal steps behind an iterator that provides access to the values within the tree. Doing this will allow the user of your iterator to process the data as they wish without concern for explicitly traversing the tree.

More specifically, you will use Python's `yield` statement to define a generator. This allows you to write your generator code in a manner very similar to a recursive depth-first traversal without concern for "going back up" the tree.

Note, carefully, that each recursive call will be returning an iterator (a generator). We can (and should) use the Python "`yield from`" when making the recursive call to `yield`, in turn, each element of the generator.

3.1 The Traversals

In your `bst.py`, you will define the following additional functions:

- `prefix_iterator` This function takes as an argument a BST and returns an iterator (using `yield`) of the elements in prefix order wherein, for a given node, the node is visited before its children (visit the left child before the right child).
- `infix_iterator` This function takes as an argument a BST and returns an iterator (using `yield`) of the elements in infix order wherein, for a given node, the node is visited after its left child but before its right child.
- `postfix_iterator` This function takes as an argument a BST and returns an iterator (using `yield`) of the elements in postfix order wherein, for a given node, the node is visited after its children (visit the left child before the right child).

3.2 Testing

In a file named `bst_iterator_tests.py`, write test cases for the functions above.

4 Testing

As mentioned in the syllabus, your code is periodically graded prior to the deadline and you will receive automated feedback based on the results of my tests. *But*, in order to receive *any* feedback, your tests much provide 100% test coverage of the code you are submitting. If you do not have 100% coverage, this is the only feedback you will receive.

100% test coverage means that every line of your code is run at some point in some test. Imagine if that weren't the case. That means that you have a line of code (or multiple lines of code) that could do anything, and you'd never know. You never tested them!

Your tests will go in files named `bst_tests.py` and `bst_iterator_tests.py`.

5 GitHub Submission

Push your finished code back to GitHub. Refer to Lab 0, as needed, to remember how to push your local code.