Lab 4, CSC 202

Here we will (1) implement more advanced and general-purpose binary search trees, and
(2) generate graphs to explore big-O characteristics of randomly generated binary search trees.

Make a Git Repository

Your repository should contain these files: **.gitignore, bst.py, bst_tests.py, bst_graphs.py**:

bst.py

```
import sys
import unittest
from typing import *
from dataclasses import dataclass
sys.setrecursionlimit(10**9)
```

bst_tests.py

```
import sys
import unittest
from typing import *
from dataclasses import dataclass
sys.setrecursionlimit(10**9)

from bst import *

class BSTTests(unittest.TestCase):
    def test_example(self):
        pass

if (__name__ == '__main__'):
    unittest.main()
```

bst_graphs.py

```
import sys
import unittest
from typing import *
from dataclasses import dataclass
import math
import matplotlib.pyplot as plt
import numpy as np
import random
sys.setrecursionlimit(10**9)

from bst import *
```

```
TREES_PER_RUN : int = 1e4

def example_graph_creation() -> None:
    # Return log-base-2 of 'x' + 5.
    def f_to_graph( x : float ) -> float:
        return math.log2( x ) + 5.0

    # here we're using "list comprehensions": more of Python's
    # syntax sugar.
    x_coords : List[float] = [ float(i) for i in range( 1, 100 ) ]
    y_coords : List[float] = [ f_to_graph( x ) for x in x_coords ]

    # Could have just used this type from the start, but I want
    # to emphasize that 'matplotlib' uses 'numpy''s specific array
    # type, which is different from the built-in Python array
    # type.
    x_numpy : np.ndarray = np.array( x_coords )
    y_numpy : np.ndarray = np.array( y_coords )

    plt.plot( x_numpy, y_numpy, label = 'log_2(x)' )
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.title("Example Graph")
    plt.grid(True)
    plt.legend() # makes the 'label's show up
    plt.show()

if (__name__ == '__main__'):
    example_graph_creation()
```

A More Advanced Binary Search Tree

A BST depends on being able to determine which of two values—say, `a` and `b`—is less than the other.

So far, we've just used the `<` operator to do this. But this is limiting for a couple of reasons:

(1) Not all data types support the `<` operator (imagine you want to store instances of some custom class in your BST).

(2) Even if the `<` operator is supported for `a` and `b`, you might not want to use it. E.g., say you're storing strings in a BST, but you want the strings to be sorted by something other than alphabetical order (which is what the `<` operator would give you).

Implement a binary search tree that does *not* use

(1) the `<` operator,

(2) the `>` operator, or

(3) other similar operators (`>=`, `<=`, `==`, `!=`).

Instead, your new BST will store a `comes_before` operation. This will literally be a field that has a *function* as a value, a function that takes in `a` and `b` and returns `True` if `a` is less than `b`; `False` otherwise.

Again, this function must take in two values and return `True` if the first should come before the second. The type annotation/hint for such a function is `Callable[[Any,Any],bool]`.

In **bst.py**, implement the following types and functions:

- A `BinTree` type that is either a `Node` or `None`. The element/value field in `Node` should be of type `Any`.
- A frozen `BinarySearchTree` class that contains two fields: a `comes_before` function (of type `Callable[[Any,Any],bool]`) and a `BinTree`.
- The following functions. Most will require helper functions that accept a `comes_before`-type function as an argument:
    - `lookup` — given a `BinarySearchTree` and a value as arguments, return `True` if the value is stored in the tree and `False` otherwise. You will want to use a helper function that operates on `BinTree`s. This helper function will need to take a `comes_before`-type function as an extra input.

      Do not use the `==` *or* `!=` *operators*. The key is to use `comes_before` any time you need to compare two values. How do you check whether two values `a` and `b` are equal? You know they are equal if `comes_before( a, b )` returns `False` and `comes_before( b, a )` also returns `False`.
    - `insert` — given a `BinarySearchTree` and a value as arguments, add the value to the tree by using the `comes_before` attribute to determine which path to take at each node; insert into the left subtree if the value "comes before" the value stored in the current node and into the right subtree otherwise. Again, you should write a helper function that does all the recursive work using `BinTree`. This helper function needs to accept the `comes_before` field of `BinarySearchTree` as another argument.

      This function returns the resulting `BinarySearchTree`.

      Make sure to avoid inserting duplicate values!
    - `delete` — given a `BinarySearchTree` and a value as arguments, remove 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 subtree come before and the values in the right subtree do not. If the tree happens to have multiple

nodes containing the value to be removed, only a single such node will be removed.

This function returns the resulting `BinarySearchTree`.

Test Cases

In `bst_tests.py`, write test cases to verify that your implementation works correctly for the following `comes_before` functions:

(1) alphabetically order strings,
(2) order 2D points by their increasing distance from the origin (prepare for this by making a simple `Point2D` class), and
(3) *reverse* order integers (e.g., treat the number 10 as less than the number 9).

3 Graphs and Report

*Do the following in **bst_graphs.py**.*

What is the height of an "average" binary search tree? How does that height increase as N increases?

First, let TREES_PER_RUN be 10,000 (written as `1e4` in the provided code). Note that all-caps is the established Python style for constant, global variables. This is a large enough number that if you generate TREES_PER_RUN different random trees that all have a given size N, you can explore the behavior of an "average" binary search tree of size N.

Write a function `random_tree` that takes in an integer `n` and generates a `BinarySearchTree` containing `n` random floats in [0,1]. (Use `random.random()` to do this.) This `BinarySearchTree` should use a `comes_before` meant for putting floats in ascending order.

Manually experiment (using `time.perf_counter()`) to find some `n_max` such that it takes 1.5-2.5 seconds to do the following TREES_PER_RUN times: generate a random tree of size `n_max`, then calculate the height of that tree.

Stated again for clarity: it should take 1.5-2.5 seconds to randomly generate and determine the height of TREES_PER_RUN binary search trees.

Graph #1: Height of Random Tree as Function of N

Make a graph where

(1) the X axis is N (the number of values in a binary search tree), and
(2) the Y axis the average height of `TREES_PER_RUN` randomly generated binary search trees, each of size N).

Use 50 different N samples spaced evenly from 1 up to the `n_max` you just came up with.

Use `matplotlib` to make your graphs directly using Python. See the example function in **bst_graphs.py**.

Graph #2: `Insert` Time as Function of N

Make a graph where

(1) the X axis is N, and
(2) the Y axis is the average time it takes to `insert` a new value into a randomly generated binary search tree of size N—averaged across TREES_PER_RUN different random BSTs of size N.

Again, use 50 evenly spaced N values from 1 up to some `n_max`—possibly a different `n_max` from what you used in the previous graph (experimentally find an `n_max` such that the total time required to generate and then insert a value into `TREES_PER_RUN` different trees takes from 1.5 to 2.5 seconds).

If it becomes too resource intensive to generate the data for either of these graphs, you are free to reduce TREES_PER_RUN.

Report

Make a document (in Word or whatever) including the two graphs along with brief explanations for what you see (along with what you should expect to see).