Lab 4, CSC 202

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

1 Setting Up Code

At this point, you should be able to create your own Git repository (hosted on the GitHub account that you've already been using, presumably). Coordinate with your group however seems best, e.g., Person A in a group can create a Git repository on their GitHub account and then Person B can access that repository.

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

bst.py

```
import sys
import unittest
from typing import *
from dataclasses import dataclass
sys.setrecursionlimit(10**6)
```

bst_tests.py

```
import sys
import unittest
from typing import *
from dataclasses import dataclass
sys.setrecursionlimit(10**6)

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
```

```python
import matplotlib.pyplot as plt
import numpy as np
import random
sys.setrecursionlimit(10**6)

from bst import *

TREES_PER_RUN : int = 10000

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()
```

2 A More Advanced Binary Search Tree

Do the following in bst.py.

You will implement a binary search tree that contains a comes_before operation—literally a field that has a function as a value. Operations on a binary search tree revolve around a "less than" relationship that determines if searching through a tree progresses to the left child or to the right child. Until now you've just used the < operator to handle this. But in this lab you will instead allow the user of your BST class to provide (at the time of creation) a function to determine if one value comes before another. 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 a file named bst.py, implement the following class and functions:

2.1 To Do

- A BinTree type that is either a Node or None. The element/value field in the Node should be of type Any.
- A frozen BinarySearchTree class that contains two fields: a comes_before function and a BinTree.
- The following functions. Most will require helper functions that accept the comes_before function as an argument (use the type Callable[[Any,Any],bool] for your type hints—this type hint describes a function or function-like object that takes two Any's as arguments and returns True/False).
  - is_empty — given a BinarySearchTree, return True if the tree is empty, False otherwise.
  - insert — given a BinarySearchTree and a value as arguments, insert adds the value to the tree by using the comes_before function to determine which path to take at each node; inserts into the left subtree if the value "comes before" the value stored in the current node and into the right subtree otherwise. You will almost certainly want to write a helper function that accepts the comes_before field of BinarySearchTree as another argument (remember the just-described type hint).

    This function returns the resulting BinarySearchTree.
  - lookup — given a BinarySearchTree and a value as arguments, return True if the value is stored in the tree and False otherwise. Note, however, that the tree was not created with an equality comparison function. Instead, you will use the comes_before function to determine if the value appears in the tree. More specifically, when comparing two values, if neither value "comes before" the other, then the values will be considered equal (i.e., for our purposes, (not (a < b) and not (b < a)) -> a = b).
  - delete — given a BinarySearchTree and a value as arguments, delete removes 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.

2.2 Test Cases

In bst_tests.py, write test cases to verify that your implementation works correctly for the following comes_before functions: basic numeric ordering of integers, alphabetic ordering of strings, and Euclidean-distance-from-zero of X/Y points (make a simple class called Point2 containing X/Y coordinates).

3 Graphs and Report

Do the following in bst_graphs.py.

3.1 Heights of Randomly Generated Trees.

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

First, let TREES_PER_RUN = 10,000. (All-caps is the known 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 (roughly speaking) can explore the behavior of an “average tree” of that N value.

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 function that relies on standard < behavior.

Experimentally find some n_max such that if you do the following TREES_PER_RUN times: generate a random tree of size n_max and then calculate its height—then the total time taken is from 1.5-2.5 seconds.

Make a graph of average tree height (y axis) as a function of N (x axis). Use 50 different N samples spaced evenly from N=0 to N=n_max. At each N you'll find the average height of TREES_PER_RUN random trees of size N.

Use matplotlib to make your graphs directly using Python. There should be example code for doing so in the repository code for this lab.

3.2 Inserting a Value into Random Trees

Make a graph that plots time-required-to-insert-a-random-value-into-an-average-tree-of-size-N (y axis) as a function of N (x axis). Again use 50 evenly spaced N values from 0 up to some n_max (possibly a different n_max from the one you used in 3.1.1—pick an n_max where the total time to generate TREES_PER_RUN trees and insert a number into each one takes from 1.5-2.5 seconds).

NOTE: If it becomes too difficult to generate the data for these graphs, you are free to reduce TREES_PER_RUN.

3.3 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).