Project 5: Binary Search Trees

Expected Duration: 8 hours
Reminder: Read through this entire project description before you begin working on it.

Introduction

Natural language processing (NLP) involves leveraging the power of computing systems to analyze natural languages (such as those we speak or write) to determine things like meaning, authorship, etc. Some of the most important advances in NLP occurred when computer scientists realized that statistics could be used to derive and predict a great deal about natural languages – even without an understanding of what is written or said. Of course every good statistic begins with the act of counting something - consequently, an important and common task in NLP is the act of counting things like letters and words. It is desirable that such counting can be performed efficiently, and the results accessed quickly. A binary search tree offers us the ability to do both. In this lab you will construct and utilize a binary search tree to compile letter frequency counts from a sample text.

In CS 1400, you may recall, you learned how to do word counts using a dictionary—this problem is similar, only we are counting letters, not words, and we are building our own key, value data structure vs using the built-in dictionary type.

Note: This is a fancy way of saying that you canNOT utilize Python's dictionary type Yes, there are a number of ways in Python to do this exercise that don't require us to write much code of our own—but a lot of high-powered databases and search engines use some form of tree structure to store and retrieve dynamic data fast, and here we get a peek under the hood. This is, after all why you're in this class.

Part 1: BST

Not surprisingly, your first task will be to create a binary search tree ADT as a class called BST. You should implement this class in a file called bst.py. Your ADT must implement the following methods:

- **size():** Return the number of nodes in the tree.
- **is_empty():** Return True if there aren't any nodes in the tree, False otherwise.
- **height()**: Return the height of the tree, defined is the length of the path from the root to its deepest leaf. A tree with zero nodes has a height of -1.
- **add(item)**: Add item to its proper place in the tree. Return the modified tree.
- **remove(item)**: Remove item from the tree if it exists, if not – do nothing. Return the resulting tree.
- **find(item)**: Return the matched item. If item is not in the tree, raise a ValueError.
- **inorder():** Return a list with the data items in order of inorder traversal.
- **preorder():** Return a list with the data items in order of preorder traversal.

- **postorder():** Return a list with the data items in order of postorder traversal.
- **[Optional] print_tree():** print the values in the tree (in any way you wish). Useful for debugging purposes

Your BST class must be generic is the sense that it can store and operate on any kind of data items that are comparable. For instance it should be possible to store a set of integers in one instance of your BST class, and a set of strings in another (without modification).

Part 2: Counting with trees

For part 2 of this project you are to use your BST to count the occurrence of letters in a sample test file (note: for our purposes we will NOT consider whitespace characters – you should ignore tabs, spaces, and newlines). To do this you will need to be able to store both a character and its current count in the tree. One way to do this is to utilize a simple class that stores both of these values, and is comparable on one of them (i.e. you can compare objects of the class using >, <, etc.). To this end we have provided you with a main.py file that contains (among other things) a class called Pair. Study this class, and use it to store your character counts in the tree.

Your main.py must implement a function make_tree, which takes no parameters but returns a tree constructed from the input file "around-the-world-in-80-days-3.txt". Apart from being convenient as you test your own code, our automated tests expect this function to be available in your main.py file.

We suggest you break this problem into the following sequence of sub-tasks:

1) Read from the file character by character. If a character is not in the tree, insert it in its proper place and set its associated count value to 1.
2) If a character IS in the tree, then retrieve the Pair object and increment the count by 1 (note: do not remove the Pair from the tree when you do this). Ignore whitespace and punctuation characters but count all others.
3) Treat upper and lowercase letters the same (i.e. 'm' is the same as 'M').
4) Organize the tree so that left characters are less than right characters.

We highly recommend that you utilize a main() function (with conditional execution) as you develop and test your project.

In a real-world scenario, we would ask various analytical problem-related questions after building the tree - but for this project we will simply examine your tree using tests in order to emphasize correctness and automated checking. Each required operation will be tested at least once.

Tips and tricks

A few things to consider for implementation:

1. The data at each node needs to account for both a letter and its count. You may use the Pair class provided in the starter code or create your own way.
2. You may NOT use a built-in dictionary as the core implementation of the tree.
3. You are welcome to implement any other operations you want that simplify the required operations.

We provide you with some test code to get you pointed in the right direction, but we also assume that you will devise your own unit tests etc. to further validate your work. At any time we may add additional tests to our test suite to ensure that your ADT functions as required. In other words – the sample tests we provide you are meant to be instructive of the kinds of tests we will run against your code – but passing these tests does not guarantee that you will receive a perfect score if you have failed to fully meet the requirements as stated in the project.

What to Submit

Submit in Canvas as `project5.zip`:

1. `main.py` - includes your make_tree() function as described above.
2. `bst.py` - your BST implementation.
3. A link to a short video showing results of your code running, about 2 minutes max. Post the video on Youtube, for example, with an unlisted link, and submit the link in comments to your submission.