**MSCF Programming Prep, 2022**

**Homework 1**

***Due At 11:59 pm EDT, Sunday, Aug. 7***

***You will lose 10 points per hour after that time***

1. **(40 points) The Binary Search Tree**
2. Start with the **BinaryTree.py** module we discussed in the Day 1 Lecture, and the **BT\_app2.py** application program. Open and examine the **BT\_app2.py** code. You will see that it is similar to the **BT\_app.py** program we discussed in the Day 1 Lecture, but creates a third, larger **BinaryTree** object. **BT\_app2.py** also has commented out sections (using the triple single quote trick) for the remaining parts of this homework.

Run **BT\_app2.py** *without making any changes* to the code and confirm that the output makes sense to you.

1. To the **BinaryTree** class in **BinaryTree.py**, add a new method **size()** that takes no argument and that returns the size of (that is, the number of values stored in) the **BinaryTree**. You may (or may not) wish to use a helper method.

In **BT\_app2.py**, comment out the triple single quotes at the beginning and end of the test code for **1.b**. Save and run **BT\_app2.py** to test that your **size()** method works correctly.

1. To the **BinaryTree** class in **BinaryTree.py**, add a new method **print\_pretty()** that takes no argument and that displays an indented representation of the current **BinaryTree** object “tipped on its left side”. (Look at the comments in the part **1.c** test code to see what we mean by this. The top node is at the left margin; the second-level nodes are indented by one tab stop from the left; the third-level nodes are indented two tab stops; and so forth.) You will probably need to use a helper method.

In **BT\_app2.py**, comment out the triple single quotes at the beginning and end of the test code for **1.c**. Save and run **BT\_app2.py** to test that your **print\_pretty()** method works correctly.

1. To the **BinaryTree** class in **BinaryTree.py**, add a new method **depth()** that takes no argument and that returns the depth of (that is, the length of the longest branch in) the **BinaryTree**. You may (or may not) wish to use a helper method.

In **BT\_app2.py**, comment out the triple single quotes at the beginning and end of the test code for **1.d**. Save and run **BT\_app2.py** to test that your **depth()** method works correctly.

1. To the **BinaryTree** class in **BinaryTree.py**, add a new method **\_\_eq\_\_()** that takes a **BinaryTree** object as argument and that returns **True** if the current **BinaryTree** object is equal to the argument **BinaryTree** object, or **False** otherwise. The **\_\_eq\_\_()** method should test whether the two **BinaryTree** objects contain the *same values*, without regard to the position of those values in the tree. That is, the two **BinaryTree** objects are “equal” if they have the same values, *even if they do not have the same shape*.

*Hint*: Since the two **BinaryTree** objects are “equal” if they have the same values, **\_\_eq\_\_()** is easy to implement by using another method that is already defined.

The **\_\_eq\_\_()** method is called when the **==** operator is used. For example, if **bt1** and **bt2** are references to **BinaryTree** objects, then **bt1 == bt2** is equivalent to **bt1.\_\_eq\_\_(bt2)**.

In **BT\_app2.py**, comment out the triple single quotes at the beginning and end of the test code for **1.e**. Save and run **BT\_app2.py** to test that your **\_\_eq\_\_()** method works correctly.

1. To the **BinaryTree** class in **BinaryTree.py**, add new methods **min()**, **max()**, and **mean()** that return the minimum, maximum, and mean values of the current **BinaryTree** object, respectively. These methods should return **None** if the current **BinaryTree** object is empty.

In **BT\_app2.py**, comment out the triple single quotes at the beginning and end of the test code for **1.f**. Save and run **BT\_app2.py** to test that your new methods works correctly.

1. **(20 points) Breadth-First Search**

The **breadth\_first\_search.py** file contains the “Pythonish pseudocode” for the breadth-first search algorithm. It also contains a **main()** function that defines **G**, the graph of nodes that we used to discuss the BFS algorithm, as a **dict** mapping each node to a list of directly connected nodes. The last statement in **main()** is a **print()** function call that should display the shortest path from node **‘Me’** to node **‘Ed’**.

1. Convert the BFS pseudocode into working Python code, that returns a list of nodes describing the shortest path from the specified root node to the target node. When you have it working, the **print()** function call at the bottom of the part 2.a test code should display:

**Shortest path from Me to Ed: ['Me', 'I', 'Q', 'Y', 'Z', 'Ed']**

1. Add more tests, to display the shortest paths from **'A'** to **'I'**, from **'B'** to **'R'**, from **'C'** to **'H'**, and from **'M'** to **'T'**. (You can simply comment out the triple single-quotes surrounding the part 2.b test code.)
2. Define a function **BFS\_con\_comps()** that: (i) takes a graph of nodes as its argument (represented as a **dict** in the form of graph **G**); (ii) uses a breadth-first search algorithm to identify all of the connected components of the graph; and finally, (iii) returns a **list** of the connected components, where each connected component is represented as a sorted **list** of nodes. You will also need to complete the definition of the **print\_con\_comps()** function, that displays the **list** of connected components in a nice way.

Comment out the triple single-quotes around the test code for part 2.c, and run the tests for the connected components of **G2** and **G**. The output produced for **G2** should be:

**graph contains 3 connected components:**

**1: ['A', 'B', 'C']**

**2: ['D']**

**3: ['E', 'F']**

1. **(40 points) Dynamic Programming**

Write your code for part 3 in a file **hw1.3.py**.

1. Our implementation of rod cutting gave the answer to, “What is the maximum revenue we can obtain from an N foot piece of rebar?” But it did not tell us the answer to, “What cuts should I make to obtain the maximum revenue from an N foot piece of rebar?” Modify the algorithm to produce the answers to *both* of these questions. Test with N ranging from 0 to 20, inclusive.
2. Implement the *weighted shortest distance* algorithm described in the Day 2 Part 2 Lecture notes. Apply it to the graph in the Day 2 Part 2 Lecture notes to verify that you get the matrix **A4** shown on slide 27, giving the shortest (least expensive) distances between all pairs of nodes in this graph when allowed to travel via *any* of the nodes 1, 2, 3, and/or 4. Display **A4**.
3. Then, apply the shortest distance algorithm to this graph:

**1**

**7**

**2**

**3**

**6**

**5**

**4**

**2**

**3**

**1**

**6**

**1**

**2**

**1**

**4**

**4**

**2**

**4**

**1**

**10**

**1**

Display the adjacency matrix **A0** and the final shortest paths matrix **A7**.

***And Finally***

***REMEMBER*** to put all team members’ names (Andrew IDs) into your source code files.Put your **BinaryTree.py**, **BT\_app2.py**, **breadth\_first\_search.py**,and **hw1.3.py** files into a **Team\_***N***\_HW1.zip** archive, where N is your team number, and upload to Canvas.