Lab 10 – CSCI 112

Due Date: Wednesday, Nov 5 at 11:59pm EST

Information

- This lab is intended to be completed **individually.**
- The files must be submitted with the exact file name provided in this file. If the file names do not match you will receive **zero** points for that file.
- Before you submit, make sure that your code runs. Any code which does not run without errors will receive **zero** points.
- Do not share your work with anyone other than Professor Khan or the TAs. You may discuss algorithms, approaches, ideas, but **NOT** exact code.
- If you submit work after a second past the due date **WILL** be locked out from submission.

Review

Heaps

Heaps are a hierarchical collection, but less rigid than a binary search tree. The only guarantee a heap has is that each node's value is smaller than either of its children. Typically, heaps are implemented using an array instead of a linked system. Math can be used to calculate a node's children or parent based on the index of the 'node'.

Assignment

Task 1 – Heaps Inheritance and Helper Functions [6 points]

The **`ArrayHeap`** we went over in class was designed as a stand-alone class, however the **`__str__`** method for heaps (and trees) is the same. Fix **`ArrayHeap`** so that it inherits from **`AbstractHeap`** and implements all the required helper methods to make the **`__str__`** method in **`AbstractHeap`** work. Make sure to remove **`__str__`** from **`ArrayHeap`**. Make sure that the heap still works by using **`testHeap.py`**.

Task 2 – Linked Heaps [9 points]

While heaps are almost always implemented as array-based trees, it is possible to implement them as a linked-based structure. There are a few things to consider when taking this approach:

1. How to find the parent of a node
2. How to find the last leaf node in the tree

Item #1 is important for when you need to "**walk up**" added items in the heap. In the **`ArrayHeap`**, we can calculate a parent based on its index, but with linked-based structures this is something we have to maintain as a pointer in the **`BSTNode`** class (Open **`bstNode.py`** and note the changes)

Item #2 is important for both adding and popping from a heap. To pop from a heap, we must swap the top value with the last leaf's value, then "**walk down**" the new value at the top of the heap until the heap is once again satisfying the properties of a heap. To add to a heap, we must know where the next open leaf spot is before we can start "walking up" the value. With **`ArrayHeaps`**, the last item or next open spot is found by index, **`len(self)-1`** or **`len(self)`**. With **`LinkedHeaps`**, we must calculate it. Luckily, because heaps are complete trees, the path can be mathematically calculated to present a series of commands ("left", "right") which will tell you how to get to the last node from the top.

The method **`_findPathToLastNode`** will return a list of strings representing the path from the root to the location of **`len(self)`**. If the **`len(self)`** is 1, the list will be empty. Otherwise, the list will contain strings (Ex: **`["left", "left", "right"]`**).

The **`add`** and **`pop`** methods for the **`LinkedHeap`** have been mostly been implemented for you. These methods use helper methods **`_walkUp`** and **`_walkDown`** which will swap values between related nodes up or down the heap as needed to preserve heap-ness. Make sure you understand the code for how the **`add`** and **`pop`** methods work.

Your task is to implement these two helper methods **`_walkUp`** and **`_walkDown`** that are used by **`add`** and **`pop`**, as well as the helper methods required for **`AbstractHeap`**. After implementing, make sure that the heap still works by using **`testHeap.py`**.

What To Turn In

Create a zip file named **Lab10_<your W&L ID>.zip**. Inside this zip archive should submit all the original files as well as the ones you created/modified.