Project 4: Heaps

Due: Friday Mar 16, 11:59 pm

1 Assignment Overview

For this project you will be implementing a heap. Rather than implementing either a min-heap or a max-heap, yours will take a comparison function that determines which of two items has a higher priority. You will then use both a min-heap and a max-heap to find the median element of a sequence.

2 Assignment Deliverables

You must turn in completed versions of the following files:

- Heap.py

Be sure to use the specified file name and to submit your files for grading via **Mimir** before the project deadline.

3 Assignment Specifications

Your task will be to complete the methods listed below:

- `__len__`
- `peek`
- `insert`
- `extract`
- `extend`
- `clear`
- `__iter__`
- `replace`
- `find_median`

Your implementation of `len`, and `peek` should run in constant time. `insert` and `extract` should run in $O(\log n)$ time. `extend` adds all of the elements in the given sequence to your heap and should run in $O(n + m)$ time, where m is the number of items in the sequence. `clear` removes all items from your heap and must run in $O(n)$ time. `iter` will enumerate all of the items in your heap and must run in $O(n)$ time. The iterator is not required to traverse the items in any particular order except that the first item *must* be the next item that would be returned by a call to `peek` or `extract`.

`replace` simultaneously inserts a new item and removes the highest priority item (including potentially the new item). It must be more efficient than calling `insert` followed by `extract`.

`find_median` determines the median (middle) item from a given sequence. If the length of the input sequence is even, you may return either of the median items arbitrarily. You should *not* compute the average of the two medians as is normal for arithmetic sequences. Your `find_median` function must be faster than sorting the sequence and finding the middle item. You must use heaps as part of your solution.

The `peek` and `extract` methods should raise an `IndexError` if the heap is empty. `replace` should *not* raise an `IndexError` because it can always return the provided item if the list is initally empty. `find_median` should also raise an `IndexError` if it is given an empty sequence. No other exceptions should be thrown.

Your heap will determine the next item by using a comparison function, `comp` supplied to the heap in its constructor. `comp(a, b)` returns `True` if the priority of `a` is greater than that of `b` and `False` otherwise. You should not use the natural ordering (that is the $<$ operator) of the elements as this will prevent you from being able to use both min-heaps and max-heaps. You should use the natural ordering within the `find_median` function. Duplicate items are allowed.

You should include comments where appropriate. In particular, you should describe the overall method used to implement your heap and your strategy for `find_median`. You must also add documentation for each of the above methods (as well as for any helper functions created) in the same style as for the pre-defined methods.

4 Assignment Notes

- Points will be deducted if your solution has warnings of any type.
- You have been supplied with stubs for the required methods. You must complete these methods to complete the project. You may add more functions than what was provided, but **you may not modify the signatures of the provided methods**.
- You do not have to worry about error checking for valid input. You may assume that the supplied reader function correctly reads the input file.
- You **do** have to worry about accessing elements from an empty heap.
- You must be able to handle duplicate elements. Duplicate elements are allowed.
- Implementations for `bool`, `is_empty`, and `repr` have been provided. Note that these rely on the `len` and `iter` functions, so you may want to complete these functions early.
- You have been provided with some unit tests that can be used to assess your solution. There are additional test cases that will be kept hidden.
- It is your responsibility to check for potential edge cases. The supplied unit tests do not account for all possible valid inputs
- The `comp` property is a function pointer that determines the ordering of objects. You can invoke it with `comp(a, b)` just like you would for a regular function. It returns `True` if `a` has a higher priority than `b`. You should not use the objects' natural ordering within the `Heap` class (but you can for the `find_median` function).
- For the `iter` method, you may want to use the `yield` keyword.
- The stub for the `find_median` function creates both a min-heap and a max-heap by using lambda expressions (anonymous functions). The `lambda` keyword is used to define a function that is not bound to a particular identifier. These lambda expressions define which of two arguments has a higher priority. Lambda expressions have been used for some the unit tests for some of the earlier projects.

- You may not modify the sequences passed to the `extend` or `find_median` functions but you may iterate over them.
- The median items of the phonetic alphabet are ‘mike’ and ‘november’.
- You may not use any of the classes in the `collections` or `heapq` modules. You *are* allowed to use the `list`, `set`, and `dict` classes and all of their member functions.