Project 6: Heaps

Due: Thursday, March 28th @ 8:00pm
This is not a team project, do not copy someone else's work.

Assignment Overview

You will be building a fixed size min heap. This means, in the constructor for the Heap class, there will be a capacity value entered and you cannot exceed the number of elements at that capacity level.

Turning It In

Be sure to submit your project as a folder named "Project6" and include in the folder:

- Heap.py, a Python3 file
- ReadMe.txt, a text file that includes:
  - Your name
  - Feedback on the project
  - How long it took to complete
  - A list of any external resources that you used, especially websites (make sure to include the URLs) and which function you used this information for. Please note that you can not use outside resources to solve the entire project, or use outside sources for multiple functions that leads to the solution of the project, then you are submitting someone else's work not yours. Outside sources use should not be used for more than one function, and it should not be more than few lines of code to solve that function. Please note that there is no penalty for using the programming codes shared by Onsay in Lectures and posted on D2L

Assignment Specifications

Node Class is completed in the skeleton file. **Do not modify this class.**
Its definition includes a key and a value. The key denotes the entity determining the ordering within the heap. The value represents the content of interest and acts as the tie breaker if the comparator keys are the same.
It's been provided with less than, greater than, equal to and string representation methods.
It also has a single accessor method to obtain the value of the Node.
Once a Node has been instantiated, the member attributes should not be modified.

Heap Class is partially completed in the skeleton file.
Method signatures and the following provided methods may **not be modified in any way.**

- __**init**__(self, capacity)
  - self._size - number of nodes currently in tree
  - self._data - built in list that stores the nodes
  - self._capacity - enforced maximum number of elements that can exist in the heap at any given time
- __**str**__(self)

- **__repr__**(self)
- **__len__**(self)

Complete and implement the following methods. **Do not modify the method signatures.**

- **_percolate_up**(self)
  - When an element initially exists in the last spot of the underlying data list, percolate it up to its valid spot in the heap representation.
  - Return: type None
  - Time Complexity: O(log(N))
  - Space Complexity: O(1)

- **_percolate_down**(self)
  - When an element initially exists in the first spot of the underlying data list, percolate it down to its valid spot in the heap representation.
  - Return: type None
  - Time Complexity: O(log(N))
  - Space Complexity: O(1)

- **_min_child**(self, idx)
  - Given an index of a node, return the index of the smaller child.
  - In the event that the index is a leaf, return -1.
  - Return: type int
  - Time Complexity: O(1)
  - Space Complexity: O(1)

- **push**(self, key, value):
  - Use the key and value parameters to add a Node to the heap.
  - In the event that pushing will exceed the limit, you must pop the smallest element out after pushing in the new element.
  - Return: type None
  - Time Complexity: O(log(N))
  - Space Complexity: O(1)

- **pop**(self)
  - Removes the smallest element from the heap.
  - Return: type same as Node.val
    - If no elements exist, return None
  - Time Complexity: O(log(N))
  - Space Complexity: O(1)

- **empty**(self)
  - Checks if the heap is empty.
  - Return: type Bool
  - Time Complexity: O(1)
  - Space Complexity: O(1)

- **top**(self)
  - Gives the root value.
    - In the case where that isn't possible, return None.

- Return: type same as Node.val
- Time Complexity: O(1)
- Space Complexity: O(1)

- **levels**(self)
- Returns all Node values on a single level into a list
- i.e. All node values in the first level go in list at index 0,
- All node values in the second level, go in list at index 1, and so on and so forth.
- Return: type List[List[Node.val]]
- If there are no nodes, return a single empty list
- Time complexity: O(N)
- Space complexity: O(N)
- You may use any list methods in this function

Application Problem

Given a list of elements, return the X most commonly occurring elements.
Example: most_x_common(['a', 'a', 'a', 'b', 'b', 'c'], 2) == {'a','b'}

- **most_x_commons**(vals, x):
- Parameters
- vals: type List[string]
- x: type int
- Return: type Set[string]
- Time Complexity: O(Nlog(x))
- N is len(vals)
- You are guaranteed that x is always <= N
- Space Complexity: O(N)
- **Using any kind of sorting algorithm will result in a zero, including built in sort.**
- In the case of ties (i.e. some elements occur the same amount of times)
- Pick the greater element, lexicographically
- Must use your heap class.
- No processing/validating should or needs to be done on the string, such as striping for whitespace, etc.
- It is allowed (and encouraged) to use a dictionary to achieve the optimal run time.

Assignment Notes

- **No use of heapq module is permitted**
- Only allowed container/collection types (or class methods) allowed are built in lists (specifically to store the underlying heap class data and within Heap.levels) and built in dictionaries in most_x_common
- As always, methods and attributes with a leading underscore should not be called outside of the class definition
- Guaranteed that no duplicate nodes (meaning same key and value) will be called to be pushed into heap
- Guaranteed that only keys of type int will be pushed
- All string methods are provided for debugging purposes and aren't used in testing for any reason.
- There are valid implementations of heaps with indexing starting at 1. However, for this assignment, its required that indexing of the underlying data list starts at 0, like the actual programming language.

- Bonus: For the Heap.levels method, if you can complete it with using O(N), from strictly 1N + O(1), memory there will be an additional 5 points added to your score. *This method has to pass all levels tests to receive this additional credit.*

Rubric

- Tests (75)
- Manual (25)
  - Time Complexity (10)
    - push, pop __ / 5
    - levels __ / 5
  - Space Complexity(10)
    - push, pop __ / 5
    - levels __ / 5
  - Space & Time for Application __/ 5
  - Bonus levels space __ / 5

Project Designed by Cyndy Ishida
ACTIONS