DSC 190 - Homework 02

Due: Wednesday, January 27

Write your solutions to the following problems by either typing them up or handwriting them on another piece of paper. Unless otherwise noted by the problem's instructions, show your work or provide some justification for your answer. Homeworks are due via Gradescope on Wednesday at 11:59 p.m.

Programming Problem 1.

In a file named `augmented_treap.py`, create a class named `AugmentedTreap` which is a treap modified to perform order statistic queries, range queries, and several other useful operations, along with a class named `TreapNode` which represents a node in a treap. Your `AugmentedTreap` should have the following methods:

- `.insert(key, priority)`: insert a new node with the given key and priority. Should take $O(h)$ time. If the key is a duplicate, raise a `ValueError`. This method should return a `TreapNode` object representing the node.
- `.delete(node)`: remove the given `TreapNode` from the treap. Should take $O(h)$ time.
- `.query(key)`: return the `TreapNode` object with the given key, if it exists; otherwise raise `ValueError`. Should take $O(h)$ time.
- `.__len__()`: Returns the number of nodes in the treap.
- `.floor(key)`: return the `TreapNode` with the largest key which is $\leq$ the given key. If there is no such key, raise `ValueError`. Should take $O(h)$ time.
- `.ceil(key)`: return the `TreapNode` with the smallest key which is $\geq$ the given key. If there is no such key, raise `ValueError`. Should take $O(h)$ time.
- `.successor(node)`: return the `TreapNode` which is the successor of the given node. If there is no such node, raise `ValueError`. Should take $O(h)$ time.
- `.query_order_statistic(k)`: Returns the node which has the kth largest key all keys in the tree. Note that $k = 1$ corresponds to the minimum. Should take $O(h)$ expected time.
- `.range_query(a, b)`: return a list of all `TreapNode` objects in the tree whose keys are in the closed interval $[a, b]$. Should take $O(kh)$ expected time.

Hint: this is a complex problem! But it is similar to what you might need to do in practice to implement an augmented data structure. You can find starter code for this problem on the course webpage.

Note that in practice you probably wouldn't use `AugmentedTreap` directly. Instead, you'd create a class `DynamicSet` which wraps the treap, abstracts away all of its details, and assigns random priorities to nodes, making it a randomized binary search tree.

Problem 1.

Let's see how efficient our treap implementation is. Create a treap by generating 100,001 random keys from a normal distribution using `keys = np.random.normal(0, 10, 100_001)`. Generate 100,001 associated random priorities with `priorities = np.random.uniform(size=100_001)`. Initialize an `AugmentedTreap` (the data structure you implemented above) and insert each key and priority one-by-one.

We will time how long it takes to find the median with a treap as compared to `np.median`. Using the `time` module or the `timeit` magic function of Jupyter Notebooks, time how long it takes to compute the median using your treap and `.query_order_statistic()`. If you are using the `time` module, repeat your timing 100 times and record the average of the timings (`timeit` automatically runs multiple times and averages for you). Next, time `np.median(keys)` using the same procedure. Report the both times, and include your timing code.