DSC 190 - Homework 03

Due: Wednesday, January 26

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 at 11:59 PM.

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, along with a class named `TreapNode` which represents a node in a treap. We will assume that duplicate keys are not permitted for simplicity.

Your `AugmentedTreap` should have the following methods and attributes:

- `.root`: the root node of the treap. Should be a `TreapNode` instance.
- `.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 return `None`. Should take $O(h)$ time.
- `.__len__()`: Returns the number of nodes in the treap.
- `.query_order_statistic(k)`: Returns the node which has the kth smallest key among all keys in the tree. Note that $k = 1$ corresponds to the minimum. Should take $O(h)$ expected time.

Your `TreapNode` should have these attributes:

- `.key`: The node's key, used to place it in the BST.
- `.priority`: Node's priority, use to place it in the heap.
- `.size`: The number of nodes in subtree rooted at this node (including the node itself).
- `.parent`: The node's parent. If this is a root node, this is `None`.
- `.left`: The node's left child; if there is none, this is `None`.
- `.right`: The node's right child; if there is none, this is `None`.

You can find starter code for this problem on the course webpage, but most of the methods will be empty. As a hint, remember that a demo notebook implementing a treap is available from lecture. You can implement the needed methods by slightly modifying the code from the demo.

Note that in practice you probably wouldn't use `AugmentedTreap` directly since it may become unbalanced. 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.

Whenever you learn a new data structure, your first question is (or should be): "OK, but when will I use this?" In most cases, you'll want to use a particular data structure if it has better performance than the alternatives for solving your specific problem. In this problem, we'll compare the performance of a treap (balanced binary search tree), heap, and a dynamic array for the problem of repeatedly updating a cumulative median of a stream of numbers.

In this scenario, we're working at the checkout of the campus Target during move-in week. Our manager – who is very into business analytics – constantly wants to know the median sale price of all purchases made so far that day. In fact, they want to be updated on the median after every n sales. So if $n = 100$, we will compute the median sale price after 100 sales, again after 200 sales, and so on. Each time we compute the median, it will be of *all* sales made that day.

You're coding this up, because you don't want to compute the median by hand. Because you've taken DSC 190, you know that a treap (or other balanced BST) might be useful here. But so may be a heap, or even a dynamic array. Which is fastest?

a) Implement the following three functions:

- `experiment_array(n, k)`: Create an empty `list`, then repeat the following k times: generate n random numbers, append them to the list one-by-one, and compute the median of all numbers inserted so far using `np.median`.
- `experiment_heap(n, k)`: Create an empty instance of `OnlineMedian` from the last homework. Then repeat the following k times: generate n random numbers, insert them into the `OnlineMedian` instance one-by-one, and ask for the median with the `.median()`.
- `experiment_treap(n, k)`: Create an empty instance of the `AugmentedTreap` class you created above. Then repeat the following k times: generate n random numbers, insert them into the treap one-by-one (with random priorities), and ask for the median with `.query_order_statistic()`.

These functions simulate our Target checkout scenario. Include your code for these functions (but you don't need to include the code defining, e.g., `OnlineMedian`).

For convenience, starter code with an implementation of `OnlineMedian` is provided on the course webpage. There is no autograder for this problem (you'll submit a picture or text of your code in a normal Gradescope assignment).

b) Now we'll consider two scenarios in which the same number of purchases are made, but where your manager wants updated with a different frequency. In Scenario 1, $n = 100$ and $k = 4000$; your manager wants updated every 100 sales. In Scenario 2, $n = 4000$ and $k = 100$; your manager is more chill and wants updated only after every 4000 sales. Time each of your functions on both scenarios and use the results to create a table like the one below:

Function	Scenario 1 (sec)	Scenario 2 (sec)
`experiment_array`	?	?
`experiment_heap`	?	?
`experiment_treap`	?	?

You should see that the treap is actually never the fastest. So when is it useful? Let's say your manager didn't want the cumulative median, but instead the median of only the 1000 most recent purchases – and now they want to be updated after every purchase. This is called the **rolling median**, and it's a useful quantity in any sort of time series analysis, such as in the analysis of stock prices.

Computing the rolling median cannot be done easily with a heap, because we not only need to insert numbers into our collection, but we also need to remove the "old" purchases when their leave our window of 1000 recent sales – and heaps do not by default support deleting elements other than root. An array, too, will be very costly, because we'll need to call `np.median` over and over (what is it's time complexity?). Instead, a treap allows us to perform insertions *and* deletions quickly, and it will win.

Treaps and balanced BSTs will also be useful any time you need a dynamic set data structure (i.e., one that supports insertion, deletion, and queries) but also has some order. For instance, treaps can support fast range queries and computation of order statistics, while hash tables (their main competition for implementing dynamic sets) do not.

Programming Problem 2.

We saw in lecture how to perform a nearest neighbor query on a KD-tree. In this problem, you'll generalize that code to performing a k-nearest neighbor query.

In a file named `knn_query.py`, implement a function named `knn_query(node, p, k)` which accepts three arguments:

- `node`: A `KDInternalNode` object representing the root of a k-d tree. (See lecture for the implementation of `KDInternalNode`.)
- `p`: A numpy array representing a query point.
- `k`: An integer representing the number of nearest neighbors to find.

Your function should return two things:

- A numpy array of distances to the kth nearest neighbors, in sorted order from smallest to largest.
- A $k \times d$ numpy array of the k nearest neighbors in order of distance to the query point. Each row of this array should represent a point. In the case of a tie (two points at the same distance), break the tie arbitrarily.

In the corner case that there are fewer than k points in the tree, simply return all of the points.

Here are some hints:

- You can keep track of the k smallest numbers in a stream of numbers by keeping a max heap and inserting each number as you go. If the max heap gets larger than k elements, pop the max – it is not one of the k smallest. You can use this idea to keep track of the k nearest neighbors as you discover them.
- You'll need to update the brute force search code to return k neighbors. To do this, you might want to use `np.argpartition`.
- Now there are two reasons that we might want to check the "other" branch: first, if there could be points over there that might be within the k nearest neighbors. But we also need to look at the other branch if we simply haven't found k points on this side.

Starter code is provided. In the starter code are two files: `knn_query.py`, where you should put your work, and `kd_tree.py`, which contains the code from lecture for building a k-d tree. You can use `kd_tree.py` to build trees for testing. You only need to submit `knn_query.py`.