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.

Solution:

```
class TreapNode:
    """A node in a treap.

    Attributes
```

```python
    ----------
    key
        The node's key, used to place it in the BST.
    priority
        Node's priority, use to place it in the heap.
    size : int
        The number of nodes in subtree rooted at this node.
    parent : Optional[TreapNode]
        The node's parent. If this is a root node, this is None.
    left : Optional[TreapNode]
        The node's left child; if there is none, this is None.
    right : Optional[TreapNode]
        The node's right child; if there is non, this is None.

    """

    def __init__(self, key, priority):
        # BEGIN PROMPT
        self.key = key
        self.priority = priority
        self.parent = None
        self.left = None
        self.right = None
        self.size = 1
        # END PROMPT

    def __repr__(self):
        """Nicely displays the node."""
        return f'{self.__class__.__name__}(key={self.key}, priority={self.priority})'
    # BEGIN REMOVE

    def is_leaf(self):
        """Returns True if this node has no children, else False."""
        return self.left is None and self.right is None

    def _update_size(self):
        self.size = 1
        if self.left is not None:
            self.size += self.left.size
        if self.right is not None:
            self.size += self.right.size
    # END REMOVE


class AugmentedTreap:
    """Half heap, half binary search tree. It's a treap!"""

    def __init__(self):
        """Create an empty treap."""
        # BEGIN PROMPT
        self.root = None
        self._size = 0
        # END PROMPT
```

```python
def delete(self, x: TreapNode):
    """Delete the node from the treap.

    Parameters
    ----------
    x : TreapNode
        The node to delete. Note that this is a TreapNode object,
        not a key. If you wish to delete a node with a specific
        key, you should query to find its node.

    Example
    -------
    >>> treap = AugmentedTreap()
    >>> _ = treap.insert(1, 2)
    >>> _ = treap.insert(5, 3)
    >>> n = treap.insert(10, -3)
    >>> treap.query(10) is n
    True
    >>> treap.delete(n)
    >>> treap.query(10) is None # .query() returns None if key not in treap
    True
    """
    # BEGIN PROMPT
    # rotate the node down until it becomes a leaf
    while not x.is_leaf():
        if x.left is not None and x.right is not None:
            if x.left.priority > x.right.priority:
                self._right_rotate(x)
            else:
                self._left_rotate(x)
        elif x.left is not None:
            self._right_rotate(x)
        elif x.right is not None:
            self._left_rotate(x)

    # the node is now a leaf and can be removed. this
    # is done by removing the reference from the node's parent
    # to this node
    p = x.parent
    if p is not None:
        if x is p.left:
            p.left = None
        else:
            p.right = None

    # when we delete x, the parent's size should decrease by one, and the
    # grandparent's size, and the great-grandparent's size, all the way to
    # the root
    p = x.parent
    while p is not None:
        p.size -= 1
        p = p.parent
```

```python
        # lastly update the tree's overall size
        self._size -= 1
        # END PROMPT

    def query(self, target):
        """Return the TreapNode with the specific key.

        Parameters
        ----------
        target
            The key to look for.

        Returns
        -------
        TreapNode or None
            The node with the specific key. Assumes that keys are unique.
            If the they key is not in the treap, return None.

        Example
        -------
        >>> treap = AugmentedTreap()
        >>> treap.insert(1, 10)
        TreapNode(key=1, priority=10)
        >>> treap.insert(5, 12)
        TreapNode(key=5, priority=12)
        >>> treap.query(5)
        TreapNode(key=5, priority=12)
        >>> treap.query(823729183721893721937) is None # key not in treap
        True

        """
        # BEGIN PROMPT
        # walk down the tree, starting at root, searching for key
        current_node = self.root
        while current_node is not None:
            if current_node.key == target:
                return current_node
            elif current_node.key < target:
                current_node = current_node.right
            else:
                current_node = current_node.left
        return None
        # END PROMPT

    def insert(self, key, priority):
        """Create a new node with given key and priority.

        Parameters
        ----------
        new_key
            The node's new key. Should be unique.
        new_priority
```

```
            The node's priority. Need not be unique.

        Returns
        -------
        TreapNode
            The new node.

        Raises
        ------
        ValueError
            If the new node's key is already in the treap.

        Example
        -------
        >>> treap = AugmentedTreap()
        >>> treap.insert(99, 100)
        TreapNode(key=99, priority=100)
        >>> _ = treap.insert(40, 2)
        >>> _ = treap.insert(99, -4)
        Traceback (most recent call last):
        ...
        ValueError: Duplicate key "99" not allowed.

        """
        # BEGIN PROMPT
        current_node = self.root
        parent = None

        # walk down the tree in search of the place to put the new key
        while current_node is not None:
            parent = current_node
            if current_node.key == key:
                raise ValueError(f'Duplicate key "{key}" not allowed.')
            if current_node.key < key:
                current_node = current_node.right
            elif current_node.key > key:
                current_node = current_node.left

            # the parent's subtree is getting one more node
            parent.size += 1

        # create the new node
        new_node = TreapNode(key=key, priority=priority)
        new_node.parent = parent
        self._size += 1

        # place it in the tree
        if parent is None:
            self.root = new_node
        elif parent.key < key:
            parent.right = new_node
        else:
            parent.left = new_node
```

```
        # the heap invariant may be broken -- rotate the node up until
        # it is once again satisfied
        while new_node != self.root and new_node.priority > new_node.parent.priority:
            if new_node.parent.left is new_node:
                self._right_rotate(new_node.parent)
            else:
                self._left_rotate(new_node.parent)

        return new_node
        # END PROMPT

    # BEGIN REMOVE
    def _right_rotate(self, x: TreapNode):
        """Rotate x down to the right."""
        u = x.left
        B = u.right
        C = x.right
        p = x.parent

        x.left = B
        if B is not None: B.parent = x

        u.right = x
        x.parent = u

        u.parent = p

        if p is None:
            self.root = u
        elif p.left is x:
            p.left = u
        else:
            p.right = u

        x._update_size()
        u._update_size()

    def _left_rotate(self, x: TreapNode):
        """Rotate x down to the left."""
        u = x.right
        A = u.left
        C = x.left
        p = x.parent

        x.right = A
        if A is not None: A.parent = x

        u.left = x
        x.parent = u

        u.parent = p
```

```python
        if p is None:
            self.root = u
        elif p.left is x:
            p.left = u
        else:
            p.right = u

        x._update_size()
        u._update_size()

    # END REMOVE
    def query_order_statistic(self, k: int):
        """Return the node whose key is kth in the sorted order of keys.

        Parameters
        ----------
        k : int
            The order statistic to return. Note that k=1 is the minimum (we
            start counting from one instead of zero).

        Returns
        -------
        TreapNode
            The treap node whose key appears kth in the ordering.

        Raises
        ------
        ValueError
            If the kth order statistic doesn't exist because k is larger than
            the number of elements in the tree.

        Example
        -------
        >>> treap = AugmentedTreap()
        >>> treap.insert(1, 20)
        TreapNode(key=1, priority=20)
        >>> treap.insert(99, 10)
        TreapNode(key=99, priority=10)
        >>> treap.insert(50, 7)
        TreapNode(key=50, priority=7)
        >>> treap.query_order_statistic(1)
        TreapNode(key=1, priority=20)
        >>> treap.query_order_statistic(2)
        TreapNode(key=50, priority=7)
        >>> treap.query_order_statistic(1000)
        Traceback (most recent call last):
        ...
        ValueError: Order statistic query out of bounds.

        """
        # BEGIN PROMPT
        current_node = self.root
        while current_node is not None:
```

```
            left_size = 0 if current_node.left is None else current_node.left.size
            current_order = left_size + 1
            if current_order == k:
                return current_node
            elif current_order < k:
                current_node = current_node.right
                k = k - current_order
            else:
                current_node = current_node.left

        raise ValueError(f'Order statistic query out of bounds.')
        # END PROMPT

    def __len__(self):
        """Return the number of keys stored in the treap.

        Example
        -------
        >>> treap = AugmentedTreap()
        >>> len(treap)
        0
        >>> _ = treap.insert(-30, 30)
        >>> len(treap)
        1
        """
        # BEGIN PROMPT
        return self._size
        # END PROMPT
```

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).

Solution:

```
def experiment_array(n, k):
    numbers = []
    for i in range(k):
        for j in range(n):
            x = np.random.sample()
            numbers.append(x)
        np.median(numbers)

def experiment_treap(n, k):
    treap = AugmentedTreap()
    for i in range(k):
        for j in range(n):
            x, priority = np.random.sample(2)
            treap.insert(x, priority)
        treap.query_order_statistic(len(treap) // 2)

def experiment_heap(n, k):
    om = OnlineMedian()
    for i in range(k):
        for j in range(n):
            x = np.random.sample()
            om.insert(x)
        om.median()
```

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`	?	?

Solution:

Here's what I see on my laptop:

Function	Scenario 1 (sec)	Scenario 2 (sec)
`experiment_array`	65	**2**
`experiment_heap`	**6**	6
`experiment_treap`	13	12

The array approach is fastest in Scenario 1 and slowest in Scenario 2 by quite a large margin. The heap approach is fastest in Scenario 2. The treap is never the fastest `: (`

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`.

Solution: First we implement a helper class to keep track of the k smallest things inserted into a container. This class wraps `MaxHeap` from a previous week:

```
class KSmallest:

    def __init__(self, k):
        self.k = k
        self.heap = MaxHeap()

    def insert(self, key):
        if len(self.heap.keys) < self.k or key < self.heap.max():
            self.heap.insert(key)

        if len(self.heap.keys) > self.k:
            self.heap.pop_max()

    def __len__(self):
        return len(self.heap)

    def as_list(self):
        return list(self.heap.keys)

    def top(self):
        return self.heap.max()
```

Next, we modify `nn_query` to perform a k-NN query. In the original implementation, we only keep track of the nearest neighbor to the query point. Now we keep track of the closest k neighbors by passing around a heap (wrapped inside of an instance of `KSmallest`). When we find a leaf node, we perform a brute-force search to find the k closest neighbors and insert them one-by-one into the instance of `KSmallest`. We only explore a branch if the distance to the boundary is less than the largest distance among the k smallest distances found so far, *or* if the number of neighbors found so far is less than k.

```
import numpy as np

import kd_tree
# BEGIN REMOVE
from ksmallest import KSmallest
# END REMOVE


# BEGIN REMOVE
def brute_force_nn_search(data, p, k=1):
    """Perform a brute force NN search.

    Parameters
    ----------
    data : ndarray
```

```
        An n x d array of n points in d dimensions.
    p : ndarray
        A d-array representing a query point.
    k : int
        The number of nearest neighbors to return. Default: 1

    Returns
    -------
    ndarray
        The k nearest neighbors of p as a k-by-d array.
    ndarray
        The distance to the k nearest neighbor.

    """
    all_distances = np.sqrt(np.sum((data - p)**2, axis=1))

    if k > len(data):
        k = len(data)

    # which distances are <= k?
    closest_ix = np.argpartition(all_distances, k-1)[:k]

    # extract the k closest points and their distances
    k_points = data[closest_ix]
    k_distances = all_distances[closest_ix]

    # lastly, we need to sort each
    sort_ix = np.argsort(k_distances)
    k_distances = k_distances[sort_ix]
    k_points = k_points[sort_ix]

    return (k_points, k_distances)

def nn_query_helper(node, p, k, k_closest):
    if isinstance(node, np.ndarray):
        candidates = brute_force_nn_search(node, p, k=k)
        for point, distance in zip(*candidates):
            # we use .tolist here because if two points have the same distance,
            # Python will fall back to comparing the points themselves, in which
            # case it will run arr_1 < arr_2; this raises an error, since the
            # truth value of this is ambiguous. But comparing Python lists works
            # as expected.
            k_closest.insert((distance, point.tolist()))
    else:
        # find the most likely branch
        if p[node.dimension] >= node.threshold:
            most_likely_branch, other_branch = node.right, node.left
        else:
            most_likely_branch, other_branch = node.left, node.right

        # compute distance to boundary
        distance_to_boundary = abs(p[node.dimension] - node.threshold)
```

```
        # find nns within most likely branch
        nn_query_helper(most_likely_branch, p, k=k, k_closest=k_closest)

        # check the other branch, but only if necessary. it will be necessary
        # if the point in the most likely branch further from the query point
        # has a distance > the distance to the boundary. It will also be
        # necessary if the number of points found in the most likely branch
        # is less than k -- in that case, we need more points!
        if distance_to_boundary < k_closest.top()[0] or len(k_closest) < k:
            nn_query_helper(other_branch, p, k=k, k_closest=k_closest)

# END REMOVE
def knn_query(node: kd_tree.KDInternalNode, p: np.ndarray, k: int=1):
    """Perform a k-nearest neighbor query on a k-d tree.

    Parameters
    ----------
    node: KDInternalNode
        The node whose subtree should be searched.
    p: np.ndarray
        The point to query.
    k: int
        The number of neighbors to return. Default: 1

    Returns
    -------
    np.ndarray
        An array of distances to the k neighbors in sorted order.
    np.ndarray
        A k-by-d ndarray containing the k nearest neighbors in order of their
        distance to the query point.

    Note
    ----
    Ties are broken arbitrarily. If k is less than the number of points in the
    tree overall, then all points are returned.

    Example
    -------
    >>> data = np.array([
    ...     [1, 2, 3],
    ...     [4, 2, 1],
    ...     [1, 1, 1],
    ...     [7, 5, 5],
    ...     [3, 2, 0]
    ... ])
    >>> p = np.array([1, 1, 0])
    >>> root = kd_tree.build_kd_tree(data) # implemented in lecture
    >>> distances, points = knn_query(root, p, k=2)
    >>> distances[0]
    1.0
    >>> np.isclose(distances[1], 2.236, atol=1e-3)
    True
```

```
>>> points
array([[1, 1, 1],
       [3, 2, 0]])

"""
# BEGIN PROMPT
k_closest = KSmallest(k)
nn_query_helper(node, p, k, k_closest)
distances, points = list(zip(*k_closest.as_list()))
distances = np.array(distances)
points = np.vstack(points)
sort_ix = np.argsort(distances)
return distances[sort_ix], points[sort_ix]
# END PROMPT
```