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 smallest key among 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, where k is the number of keys in the interval $[a, b]$.

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.

Solution:

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

    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 : 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):
        self.key = key
        self.priority = priority
        self.parent = None
        self.left = None
        self.right = None
        self.size = 1

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

    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


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

    def __init__(self):
        """Create an empty treap."""
        self.root = None
        self._size = 0

    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.
```

```
        """
        # 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

        self._size -= 1

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

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

        Returns
        -------
        TreapNode
            The node with the specific key. Assumes that keys are unique.
        Example
        -------
        >>> treap = Treap()
        >>> 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)

        """
        # 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

    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.

        """
        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

    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

        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 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 = Treap()
        >>> 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)
        """
        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.')
```

```
    def __len__(self):
        return self._size

    def successor(self, x: TreapNode):
        """Find a node's successor (the next largest node by key).

        Parameters
        ----------
        x : TreapNode
            The node whose successor will be found.
        Returns
        -------
        TreapNode
            The successor of x.
        Raises
        ------
        ValueError
            If x has no successor.
        Example
        -------
        >>> treap = Treap()
        >>> x = treap.insert(3, 10)
        >>> treap.insert(6, 2)
        TreapNode(key=6, priority=2)
        >>> treap.insert(5, 12)
        TreapNode(key=5, priority=12)
        >>> treap.successor(x)
        TreapNode(key=5, priority=12)
        """
        if x.right is not None:
            return self._min_in_subtree(x.right)
        else:
            # walk up the tree until you find a node that is a left child
            while x is not None and x is x.parent.right:
                x = x.parent
            return x.parent

        raise ValueError(f'There is no successor of {x}')

    def _min_in_subtree(self, x):
        parent = x.parent
        while x is not None:
            parent = x
            x = x.left
        return parent

    def floor(self, key):
        """Find greatest node whose key is <= given key.
        Parameters
        ----------
        key
            The key whose floor will be found.
        Returns
```

```
        -------
        TreapNode
            The node whose key is the floor of the given key.
        Raises
        ------
        ValueError
            If there is no key in the tree.
        Example
        -------
        >>> treap = Treap()
        >>> treap.insert(5, 10)
        TreapNode(key=5, priority=10)
        >>> treap.insert(2, 12)
        TreapNode(key=2, priority=12)
        >>> treap.floor(10)
        TreapNode(key=5, priority=10)
        >>> treap.floor(4)
        TreapNode(key=2, priority=12)
        """
        current_node = self.root
        current_floor = None
        while current_node is not None:
            if current_node.key == key:
                return current_node
            elif current_node.key < key:
                current_floor = current_node
                current_node = current_node.right
            else:
                current_node = current_node.left

        if current_floor is None:
            raise ValueError(f'{key} has no floor.')

        return current_floor

    def ceil(self, key):
        """Find greatest node whose key is <= given key.
        Parameters
        ----------
        key
            The key whose ceil will be found.
        Returns
        -------
        TreapNode
            The node whose key is the ceil of the given key.
        Raises
        ------
        ValueError
            If there is no key in the tree.
        Example
        -------
        >>> treap = Treap()
        >>> treap.insert(5, 10)
```

```
        TreapNode(key=5, priority=10)
        >>> treap.insert(2, 12)
        TreapNode(key=2, priority=12)
        >>> treap.ceil(1)
        TreapNode(key=2, priority=12)
        >>> treap.ceil(4)
        TreapNode(key=5, priority=10)
        """
        current_node = self.root
        current_ceil = None
        while current_node is not None:
            if current_node.key == key:
                return current_node
            elif current_node.key > key:
                current_ceil = current_node
                current_node = current_node.left
            else:
                current_node = current_node.right

        if current_ceil is None:
            raise ValueError(f'{key} has no ceiling.')

        return current_ceil

    def range_query(self, a, b):
        """Returns all nodes whose keys are within [a, b].
        Parameters
        ----------
        a, b : float
            The endpoints of the interval.
        Returns
        -------
        List[TreapNode]
            A list of TreapNodes whose keys are in the interval.
        Example
        -------
        >>> treap = Treap()
        >>> treap.insert(5, 10)
        TreapNode(key=5, priority=10)
        >>> treap.insert(2, 12)
        TreapNode(key=2, priority=12)
        >>> treap.insert(7, 3)
        TreapNode(key=7, priority=3)
        >>> treap.range_query(1, 6)
        [TreapNode(key=2, priority=12), TreapNode(key=5, priority=10)]
        """
        current_node = self.ceil(a)
        result = []
        while current_node.key <= b:
            result.append(current_node)
            current_node = self.successor(current_node)
        return result
```

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.