DSC 190 - Homework 02

Due: Monday, October 16

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 `bst_with_fcd.py`, create a class named `BinarySearchTree` which implements a binary search tree with the following methods and attributes:

- `.root`: the root node of the tree; a Node object. If the tree is empty, this should be `None`.
- `.insert(key)`: insert a new node with the given key. Should take $O(h)$ time. Should allow duplicate keys. This method should return a `Node` object representing the node.
- `.delete(node)`: remove the given `Node` from the BST. Should take $O(h)$ time.
- `.query(key)`: return the `Node` object with the given key, if it exists; otherwise raise `ValueError`. Should take $O(h)$ time.
- `.floor(key)`: return the `Node` 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 `Node` with the smallest key which is $\geq$ the given key. If there is no such key, raise `ValueError`. Should take $O(h)$ time.

You can find starter code for this problem on the course webpage.

Solution:

```
class Node:
    """A node in a BinarySearchTree. Has a key."""

    def __init__(self, key, parent=None):
        self.key = key
        self.parent = parent
        self.left = None
        self.right = None


class BinarySearchTree:
    def __init__(self):
        self.root = None

    def _right_rotate(self, x: Node):
        """Rotates a node down to the right.

        Parameters
        ----------
        x : Node
            A node in the tree.

        """
```

```
        if x is None:
            return

        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

    def _left_rotate(self, x: Node):
        """Rotates a node down to the left.

        Parameters
        ----------
        x : Node
            A node in the tree.

        """
        # BEGIN PROMPT
        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
```

```
        # END PROMPT

    def insert(self, new_key) -> Node:
        """Inserts a new key into the BST.

        Parameters
        ----------
        new_key
            A node with this key will be created.

        Returns
        -------
        Node
            The created node, in case you'd like to hang on to it.

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

        while current_node is not None:
            parent = current_node
            if current_node.key <= new_key:
                current_node = current_node.right
            elif current_node.key > new_key:
                current_node = current_node.left

        new_node = Node(key=new_key, parent=parent)
        if parent is None:
            self.root = new_node
        elif parent.key <= new_key:
            parent.right = new_node
        else:
            parent.left = new_node

        return new_node
        # END PROMPT

    def query(self, target) -> Node:
        """Check whether a node exists with the given target key.

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

        Returns
        -------
        Node
            If the target is in the BST, a node with the target key is returned.
            If two nodes in the BST have the target key, either may be returned.
            If the target is not in the BST, a ValueError should be raised.
```

```
        Raises
        ------
        ValueError
            If there is no node in the BST with the target 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
        raise ValueError("Key not in BST.")

    def floor(self, target) -> Node:
        """Find the node with the largest key <= target.

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

        Returns
        -------
        Node
            A node in the BST with the largest key that is <= target.

        Raises
        ------
        ValueError
            If there is no node in the BST with a key <= target.

        Example
        -------

        >>> bst = BinarySearchTree()
        >>> _ = bst.insert(10)
        >>> _ = bst.insert(20)
        >>> _ = bst.insert(30)
        >>> bst.floor(42).key
        30
        >>> bst.floor(20).key
        20
        >>> bst.floor(5)
        Traceback (most recent call last):
          ...
        ValueError: 5 has no floor.

        """
        # BEGIN PROMPT
        current_node = self.root
```

```
        current_floor = None
        while current_node is not None:
            if current_node.key == target:
                return current_node
            elif current_node.key < target:
                current_floor = current_node
                current_node = current_node.right
            else:
                current_node = current_node.left

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

        return current_floor
        # END PROMPT

    def ceil(self, target) -> Node:
        """Find the node with the smallest key >= target.

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

        Returns
        -------
        Node
            A node in the BST with the largest key that is <= target.

        Raises
        ------
        ValueError
            If there is no node in the BST with a key <= target.

        Example
        -------

        >>> bst = BinarySearchTree()
        >>> _ = bst.insert(10)
        >>> _ = bst.insert(20)
        >>> _ = bst.insert(30)
        >>> bst.ceil(5).key
        10
        >>> bst.ceil(20).key
        20
        >>> bst.ceil(42)
        Traceback (most recent call last):
          ...
        ValueError: 42 has no ceiling.

        """
        # BEGIN PROMPT
        current_node = self.root
```

```
        current_ceil = None
        while current_node is not None:
            if current_node.key == target:
                return current_node
            elif current_node.key > target:
                current_ceil = current_node
                current_node = current_node.left
            else:
                current_node = current_node.right

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

        return current_ceil
        # END PROMPT

    def delete(self, x: Node):
        """Remove the node from the BST.

        Note that you must specify the Node, and not the key. This is more
        efficient, because specifying the key would require a query to find the
        node. If you want to delete a specific key, use .query to find the node
        first, then use this method.

        Parameters
        ----------
        x : Node
            A node in the BST. We will assume that x is indeed in the BST.

        Example
        -------

        >>> bst = BinarySearchTree()
        >>> n1 = bst.insert(10)
        >>> n2 = bst.insert(-20)
        >>> n3 = bst.insert(3.14)
        >>> bst.query(3.14) == n3
        True
        >>> bst.delete(n3)
        >>> # the next line should raise a ValueError, because .query raises
        >>> # one if the key is not in the BST
        >>> bst.query(3.14)
        Traceback (most recent call last):
          ...
        ValueError: Key not in BST.

        """
        # BEGIN PROMPT
        # rotate x down to the bottom of the tree so that it becomes a leaf
        while not is_leaf(x):
            if x.left is not None:
                self._right_rotate(x)
            else:
```

```
                self._left_rotate(x)

        parent = x.parent

        # x is now guaranteed to be a leaf. if it also has no parent, then
        # it must be that it's the only node in the tree (and therefore the root)
        # we'll need to update the tree's root, but then we're done
        if parent is None:
            self.root = None
            return

        # otherwise, x is not the root and it has a parent node. we'll need to
        # update it's parent's left/right child pointers
        if x == parent.left:
            parent.left = None
        elif x == parent.right:
            parent.right = None
        # END PROMPT


# BEGIN REMOVE
def is_leaf(x: Node):
    return x.left is None and x.right is None
# END REMOVE
```

Programming Problem 2.

In a file named `min_heap.py`, implement a `MinHeap` class. Your class should have the following methods:

- `.min()`: return (but do not remove) minimum key. If the heap is empty, this should return `nan`.
- `.decrease_key(i, key)`: reduce the value of node i's key to `key`. Raise a `ValueError` if the new key is $<$ the old key.
- `.insert(key)`: insert a new key, maintaining the heap invariant
- `.pop_min_key()`: remove and return the minimum key. If there are no elements currently in the heap, raise an `IndexError`.

Your methods should have all of the same time complexities as the corresponding methods on `MaxHeap` as discussed in lecture. You may assume that the keys are numbers.

Hint: a max heap was implemented in lecture. One solution is to modify its code to transform it into a min heap. There's another, cleverer approach. Is there an easy way to use a max heap to implement a min heap without changing the code of the max heap?

Solution: Here's the straightforward implementation of a min heap. It is a modification of the code for a max heap that was given in lecture.

```
# BEGIN REMOVE
def parent(ix):
    return (ix - 1) // 2

def left_child(ix):
    return 2*ix + 1
```

```
def right_child(ix):
    return 2*ix + 2
# END REMOVE


class MinHeap:

    def __init__(self):
        # BEGIN PROMPT
        self.keys = []
        # END PROMPT

    # BEGIN REMOVE
    def _swap(self, i, j):
        self.keys[i], self.keys[j] = self.keys[j], self.keys[i]

    def _push_down(self, i):
        left = left_child(i)
        right = right_child(i)
        if left < len(self.keys) and self.keys[left] < self.keys[i]:
            smallest = left
        else:
            smallest = i

        if right < len(self.keys) and self.keys[right] < self.keys[smallest]:
            smallest = right

        if smallest != i:
            self._swap(i, smallest)
            self._push_down(smallest)

    # END REMOVE
    def insert(self, key):
        """Insert a key into the heap."""
        # BEGIN PROMPT
        self.keys.append(key)
        self.decrease_key(len(self.keys)-1, key)
        # END PROMPT

    def min(self):
        """Return, but do not remove the minimum. If the heap is empty,
        NaN is returned.

        Example
        -------
        >>> h = MinHeap()
        >>> h.min()
        nan
        >>> h.insert(40)
        >>> h.insert(-3.14)
        >>> h.insert(20)
        >>> h.min()
        -3.14
```

```
        """
        # BEGIN PROMPT
        try:
            return self.keys[0]
        except IndexError:
            return float('NaN')
        # END PROMPT

    def decrease_key(self, ix, key):
        """Decrease the value of element i's key.

        Parameters
        ----------
        ix : int
            The index of the element whose key will be decreased.
        key
            The element's new key. This should be <= the old key.

        Raises
        ------
        ValueError
            If the element's new key is not <= the old key.

        Example
        -------
        >>> h = MinHeap()
        >>> h.insert(10)
        >>> h.insert(20)
        >>> h.insert(30) # this is element 2
        >>> h.decrease_key(2, 5)
        >>> h.min()
        5

        """
        # BEGIN PROMPT
        if key > self.keys[ix]:
            raise ValueError('New key is larger.')

        self.keys[ix] = key
        while parent(ix) >= 0 and self.keys[parent(ix)] > key:
            self._swap(ix, parent(ix))
            ix = parent(ix)
        # END PROMPT

    def pop_min_key(self):
        """Remove and return the minimum key.

        Raises
        ------
        IndexError
            If the heap is empty.
```

```
        Examples
        --------
        >>> h = MinHeap()
        >>> h.insert(30)
        >>> h.insert(10)
        >>> h.insert(20)
        >>> h.pop_min_key()
        10
        >>> h.pop_min_key()
        20
        >>> h.pop_min_key()
        30
        >>> h.pop_min_key()
        Traceback (most recent call last):
        ...
        IndexError: Heap is empty.


        """
        # BEGIN PROMPT
        if len(self.keys) == 0:
            raise IndexError('Heap is empty.')
        lowest = self.min()
        self.keys[0] = self.keys[-1]
        self.keys.pop()
        self._push_down(0)
        return lowest
        # END PROMPT
```

A clever alternative is to implement a min heap by wrapping a max heap and negating keys as you insert them.

Programming Problem 3.

In a file named `online_median.py`, create a class named `OnlineMedian` which stores a collection of numbers and has which has two methods: `.insert(x)`, which inserts a number into the collection in $O(\log n)$ time, and `.median()` which computes the median of all numbers inserted so far in $\Theta(1)$ time.

If no numbers have been inserted so far, `.median()` should return `NaN`.

Solution: As discussed in lecture, we can implement this with a min heap and a max heap. The min heap stores the upper half of the data, while the max heap stores the lower half.

As we insert a key, we need to compare it to the tops of the heaps. If it is smaller than the max of the lower heap, we insert into it. Otherwise, we insert into the upper heap.

If the heaps become unbalanced we should rebalance them. This is as simple as popping an element from the larger heap and inserting it into the smaller heap.

The following code assumes that you've also copied the implementations of `MaxHeap` and `MinHeap` into the same file.

```
# BEGIN REMOVE
def parent(ix):
    return (ix - 1) // 2
```

```
def left_child(ix):
    return 2*ix + 1

def right_child(ix):
    return 2*ix + 2

class MaxHeap:

    def __init__(self, keys=None):
        if keys is None:
            keys = []
        self.keys = keys

    def max(self):
        return self.keys[0]

    def _swap(self, i, j):
        self.keys[i], self.keys[j] = self.keys[j], self.keys[i]

    def increase_key(self, ix, key):
        if key < self.keys[ix]:
            raise ValueError('New key is smaller.')

        self.keys[ix] = key
        while parent(ix) >= 0 and self.keys[parent(ix)] < key:
            self._swap(ix, parent(ix))
            ix = parent(ix)

    def insert(self, key):
        self.keys.append(key)
        self.increase_key(len(self.keys)-1, key)

    def pop_max(self):
        if len(self.keys) == 0:
            raise IndexError('Heap is empty.')
        highest = self.max()
        self.keys[0] = self.keys[-1]
        self.keys.pop()
        self._push_down(0)
        return highest

    def _push_down(self, i):
        left = left_child(i)
        right = right_child(i)
        if left < len(self.keys) and self.keys[left] > self.keys[i]:
            largest = left
        else:
            largest = i

        if right < len(self.keys) and self.keys[right] > self.keys[largest]:
            largest = right

        if largest != i:
```

```
            self._swap(i, largest)
            self._push_down(largest)

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


class MinHeap:

    def __init__(self, keys=None):
        self._max_heap = MaxHeap(keys)

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

    def min(self):
        return -self._max_heap.max()

    def _get_key(self, ix):
        return -self._max_heap.keys[ix]

    def decrease_key(self, ix, key):
        if key > self._get_key(ix):
            raise ValueError('New key is larger.')

        self._max_heap.increase_key(ix, -key)

    def insert(self, key):
        self._max_heap.insert(-key)

    def pop_min(self):
        return -self._max_heap.pop_max()
# END REMOVE

class OnlineMedian:

    # BEGIN REMOVE
    def __init__(self):
        self._upper = MinHeap()
        self._lower = MaxHeap()
        self._empty = True
    # END REMOVE

    def insert(self, key):
        """Insert a key into the collection."""
        # BEGIN PROMPT
        if self._upper:
            if key >= self._upper.min():
                self._upper.insert(key)
            else:
                self._lower.insert(key)
        else:
            self._lower.insert(key)
```

```
        if len(self._lower) - len(self._upper) >= 2:
            self._rebalance(self._upper, self._lower)
        elif len(self._upper) - len(self._lower) >= 2:
            self._rebalance(self._lower, self._upper)

        self._empty = False
        # END PROMPT

    def median(self):
        """Return a median of the collection.

        If the collection is empty, returns nan.

        Example
        -------

        >>> om = OnlineMedian()
        >>> om.median()
        nan
        >>> om.insert(30)
        >>> om.insert(10)
        >>> om.insert(-20)
        >>> om.median()
        10

        """
        # BEGIN PROMPT
        if self._empty:
            return float('nan')

        if len(self._lower) == len(self._upper):
            return self._lower.max()
        elif len(self._upper) > len(self._lower):
            return self._upper.min()
        else:
            return self._lower.max()
        # END PROMPT

    # BEGIN REMOVE
    def _rebalance(self, smaller, larger):

        def pop(container):
            if isinstance(container, MinHeap):
                return container.pop_min()
            else:
                return container.pop_max()

        while len(larger) - len(smaller) >= 2:
            x = pop(larger)
            smaller.insert(x)
    # END REMOVE
```

Problem 1. (Challenge)

Note: this problem is *not graded*. It's just a problem that you can think about it you're interested. Feel free to come to office hours to discuss it.

How would you design a class that maintains a collection of numbers and supports the following operations:

- `.insert(x)`: insert a new number into the collection;
- `.remove(x)`: remove a number from the collection;
- `.min_gap()`: return the minimum gap between any two numbers in the collection.

There is a solution that has $O(\log n)$ time, where n is the size of the collection. The solution makes use of balanced binary search tree(s).

Solution: The idea is to keep *two* binary search trees – one for the numbers in the collection, and one for the pairwise gaps between consecutive elements. Call the first BST the *elements* BST and the second the *gaps* BST. We will assume that all BSTs are self-balancing so that, e.g., query, insertion, and deletion take $O(\log n)$ time.

When we insert a new element x into the collection, we first find its floor and ceiling in the elements collection; let these be u and v respectively. Inserting x will "break" the gap between u and v into a gap of size $x - u$ and $v - x$. First, we remove $v - u$ from the gaps BST. Then we insert $x - u$ and $v - x$. Lastly, we add x to the elements BST. This is all that needs to be done for insertion.

For removal, we do much the same as above, except now removing x will replace two gaps with one larger gap. We first find the floor and ceiling of x; call these u and v. We remove $x - u$ and $v - x$ from the gaps BST and insert a gap of size $v - u$. We then remove x from the elements BST.

`.min_gap()` is implemented by finding the minimum element of the gaps BST, which can be done in logarithmic time.

Since we've assumed a self-balancing BST, all of the above operations take logarithmic time in the worst case.