HW 1: Project Report

Name: Detravious Jamari Brinkley

UFID: 13363139

UF Email: dj.brinkley@ufl.edu

This project entails building a seat booking service in python3 for Gator Events. Users can efficiently manage seat reservations by reserving, canceling, and performing other actions detailed below. I use a binary min-heap to manage available seats and waitlist seats. I use a red-Black tree to mange the seats that are reserved. Both data structures and their operations are implemented in my code and displayed below. This was a cool project. It gave me a base of implementing previously mentioned data structures and base code to if I wanted to build a system as such.

To run Gator Ticket Master on any test file, use either command below

```
python3 gatorTicketMaster.py testCase1.txt > testCase1_output_file.txt
python3 gatorTicketMaster.py testCase2.txt > testCase2_output_file.txt
python3 gatorTicketMaster.py testCase3.txt > testCase3_output_file.txt
python3 gatorTicketMaster.py testCase4.txt > testCase4_output_file.txt
python3 gatorTicketMaster.py testCase5.txt > testCase5_output_file.txt

make PROGRAM_NAME=gatorTicketMaster TEST_FILE=testCase1.txt
make PROGRAM_NAME=gatorTicketMaster TEST_FILE=testCase2.txt
make PROGRAM_NAME=gatorTicketMaster TEST_FILE=testCase3.txt
make PROGRAM_NAME=gatorTicketMaster TEST_FILE=testCase4.txt
make PROGRAM_NAME=gatorTicketMaster TEST_FILE=testCase5.txt
```

seat_booking_script.py

```
import heap_operations, tree_operations, visualizations, time


class SeatBooking():
    """A class to manage seat creation, reservations, cancelations, and more."""
    def __init__(self):
    [code]

      def initialize(self, seat_count: int):
        """Initialize the events with the specified number of seats, denoted as “seatCount”.
        The seat numbers will be sequentially assigned as [1, 2, 3, ..., seatCount] and added to the list of unassigned seats.

        Parameter:
        ----------
        seat_count: `int`
            The #seats to be initally unassigned

        """
    def available(self):
        """Print the number of seats that are currently available for reservation and the length of the waitlist."""


    def reserve(self, user_id: int, user_priority: int):
        """Allow a user to reserve the seat that is available from the unassigned seat list and update the reserved seats tree. If no seats are currently available, create a new entry in the waitlist heap as per the user’s priority and timestamp. Print out the seat number if a seat is assigned. If the us
```

```
er is added to the waitlist, print out a message to the user stating that he is added to the waitlist.

        Parameters:
        -----------
        user_id: `int`
            The user that wants to reserve a seat

        user_priority: `int`
            The priority of the user that want wants to reserve a seat
        """

    def cancel(self, seat_id: int, user_id: int):
        """Reassign the seat to a user from the waitlist heap. If the waitlist is empty, delete the node and add it back to the available seats.

        Parameters:
        -----------
        seat_id: `int`
            The seat to remove for user and to insert into available seats heap

        user_id: `int`
            The user that wants to reserve a seat

        """
    def exit_waitlist(self, user_id: int):
        """If the user is in the waiting list, remove him from the waiting list. If the user is already assigned a seat prior to this, the user must use the cancel function to cancel his reservation instead.

        Parameter:
        ----------
        user_id: `int`
            The user that wants to exit the waitlist heap
        """

    def update_priority(self, user_id: int, user_priority: int):
        """Modify the user priority only if the user is in the waitlist heap. Update the heap with this modification.

        Parameters:
        -----------
        user_id: `int`
            The user that wants to reserve a seat

        user_priority: `int`
            The new priority level to assign to user

        """

    def add_seats(self, counts: int):
        """We add the new seat numbers to the available seat list. The new seat numbers should follow the previously available range.

        Parameter:
        ----------
        counts: `int`
            The total #new seats to add to available seats list
        """
    def print_reservations(self):
        """Print reservations"""

    def release_seats(self, user_id_1: int, user_id_2: int):
        """
        Release all the seats assigned (in Red-Black Tree) to
```

```
users whose IDs fall
        in the range [user_id_1, user_id_2]. It is guaranteed
that user_id_2 >= user_id_1.
        Remove users from the waitlist if they are present th
ere. The status of the change
        should be printed ordered by user IDs in the range.

        Once removed from the RBT, insert seats back into the
seat heap to make them available.
        """


    def quit(self):
        """Anything below this command in the input file will
not be processed. The program terminates either when the quit
command is read by the system or when it reaches the end of t
he input commands, which ever happens first."""



    def get_highest_priority_user(self, waitlist: list):
        """Helper function to find the user with the highest
priority and earliest timestamp."""

      def assign_higest_priority_user(self):
        """Helper function to assign the user with the highes
t priority and earliest timestamp and to perform neccessary a
ctions."""
```

`heap_operations.py`

```
from abc import ABC, abstractmethod

class HeapFactory(ABC):
    """To create heap objects and perform heap operations. Ea
ch function below will contain either one or both parameters.
```

```
    Parameters:
    -----------
    i: `int`
        The index of a node

    key: `int`
        The value of a node

    """

    def __init__(self):

    def get_root_node(self, i: int):
                """Get root node of heap"""

    def get_left_node(self, i: int):
            """Get left child node of heap"""

    def get_right_node(self, i: int):
            """Get right child node of heap"""

    # @abstractmethod
    def insert(self, key: int):
        """Insert a new key (priority, time_reserved, user_id) into the min-heap."""

    def decrease_heap_key(self, i: int, key: int):
        """Decrease the key value at index i to the new key, maintaining the min-heap property."""

    # @abstractmethod
    def extract_min(self):
              """Extract root node as it's the min node."""

    def delete_node(self, key: int):
        """Delete any key in min-heap"""
    def build_heap(self):
        """Take an unordered list of n elements to place into
a heap data structure.
        O(log n): Call min_heapify() which worse case travers
es the height of the heap log_2 (n) to restore heap property
        O(n): For every element, we call min_heapify()
        """

    def search(self, key):
        """Search for a key (user) in the heap."""

    def min_heapify(self, i: int):
        """To maintain the min-heap property of the parent no
de being smaller than the child node. If the parent node is l
arger than either of its children, the function "sinks" the p
arent down the tree, swapping it with the smallest child, unt
il the heap property is restored.

        O(log n): Call min_heapify() which worse case travers
es the height of the heap log_2 (n) to restore heap property
        O(i=1): For element i, call min_heapify()
        """

class SeatHeap(HeapFactory):
    """Inherit functions from the HeapFactory class. Use this
class to store available seats and to perform other operation
s that are in functions below."""

    def build_heap(self, available_seats: list):
            """Take an unordered list of n elements to place
into a heap data structure."""

    def extract_min(self):
        """Extract root node as it's the min node."""
    def insert(self, seat_id: int):
        """Insert a new seat_id into the min-heap without pri
ority."""

class WaitlistHeap(HeapFactory):
    """Inherit functions from the HeapFactory class. Use this
class to store users in waitlist and to perform other operati
ons that are in functions below.."""

    def build_heap(self, waitlist_seats: list):
            """Take an unordered list of n elements to place
into a heap data structure."""

    def insert(self, key: int):
        """Insert a new key (priority, time_reserved, user_i
d) into the min-heap."""

    def decrease_heap_key(self, i: int, key: int):
        """Decrease the key value at index i to the new key,
maintaining the min-heap property."""
```

`tree_operations.py`

```
from abc import ABC, abstractmethod

class Node:
    def __init__(self, key=None, value=None, color="red", lef
t=None, right=None, parent=None):
        self.key = key          # The user ID
        self.value = value      # The seat ID
        self.color = color      # Used for red-black trees
        self.parent = parent    # Reference to the parent nod
e
        self.left = left        # Reference to the left child
        self.right = right      # Reference to the right chil
```

```
d

    def __repr__(self):
        """Custom string representation for the node"""

class TreeFactory(ABC):
    """To create tree objects and perform tree operations"""

    def __init__(self):

    @abstractmethod
    def insert(self, key: int, value: int):
            """Insert new node and ensure properties are main
tained"""

    def rebalance(self, new_node: int):
            """Rebalance to ensure properties are maintaine
d"""

    def search(self, key: int):
        """Search for a node with the given key in the Red-Bl
ack Tree."""

    def _search_recursive(self, node: int, key: int):
        """Helper function that recursively searches for the
node with the given key."""

    def delete(self, key: int):
        """Delete a node with the specified key from the Red-
Black Tree."""

    def _transplant(self, u: int, v: int):
        """Replace the subtree rooted at u with the subtree r
ooted at v."""

    def _minimum(self, node: int):
        """Find the minimum node in a subtree rooted at node."""

    def _delete_fix(self, x: int):
        """Fix the Red-Black Tree properties after deletion."""

class RedBlackTree(TreeFactory):
    """Inherit functions from the HeapFactory class. Use this class to implement a Red-Black Tree"""

    def __init__(self):

    def insert(self, key: int, value: int):
        """Insert a node with the specified key and value into the Red-Black Tree"""

    def rebalance(self, new_node: int):
            """Rebalance to ensure properties are maintained"""

    def left_rotate(self, current_node: int):
        """Perform a left rotation on the given node in a Red-Black Tree to maintain properties."""

    def right_rotate(self, parent_node: int):
            """Perform a right rotation on the given node in a Red-Black Tree to maintain properties."""
```