Name: Shahid Shareef Mohammad
UFID: 84381774

The **gator ticketmaster** project contains **main.py, rbtree.py,min_heap.py**:

- main.py contains the main application logic, processing commands like reservation, cancellation, and updates. This file imports the Red-Black Tree and Min-Heap modules to handle core functionality.
- rbtree.py contains the implementation of redblack trees built in consideration of seat_id for building the tree, which are utilised in maintain the **reservations** along with its functionalities. Contains insert, delete, search,inorder traversal and balancing functionalities.
- The min_heap.py implements a min heap(for maintaining the **available seat**s) and a max heap(to maintain the **waitlist(priority queue)** where the user with higher priority number needs to pop. contains insert, pop, remove,heapify functionalities.
- The min_heap.py contains functionalities where the **available seats** will only be having one element entries(seat_id), but the **waitlis**t has entries in the form of [user_id,priority_value].(function names ending with _single in min heap are only utilised by available seats and functions ending with _multi are only utilised by waitlist(priorityqueue), the rest are common)

By separating the code into different files/modules the project maintains clear organization and readability, also making it easy to modify individual components without impacting the entire codebase. main.py imports classes from both rbtree.py and min-heap.py

rbtree.py →main.py ← min-heap.py
*waitlist = priorityqueue, *available seats = minheap, *redblacktree = reservation
Data Flow
In main.py we got *main* function which will be taking in the test_file name from the command line and reading through the commands in that test_file. It parses the commands in a way using regex, to separate the actual command and the parameters in the *process_command* function. After segregating based on the command, the respective function will be called which were declared in the *GatorTicketMaster* class in main.py. We also declare the output file name dynamically based on the input file name which inturn is captured from CLI.

Functions in *GatorTicketMaster:*

<u>Note:</u> gatorticketmaster initiates empty redblacktree for maintaining reservations, empty available_seats min heap for tracking the seats, and an empty priority queue for maintaining waitlist. A variable to store the output file name which is shared from main.and another variable started to check whether the ticket system is initiated or not.

Classes/data structures in respective files

rbtree→*[Redblacktree()--for reservations storing the entries by seat_id*
Minheap→*[min-heap()*--for implementation of priority queue and availableseats*,*
class priorityqueue()--to maintain waitlists, class availableseats()--to maintain the seats]
Main→*[class gatorticktemaster()--co-ordinates reservation,priority waitlist and available seats, def main()]*

Function prototypes

1)*initialize(count)*: the available seats min_heap will be initiated with seats ranging from [1 to count].

```
def initialize(self, seat_count):
       if not self.started:
           self.available_seats.adding(seat_count)
           self.output_file.write(f"{seat_count} Seats are made available for 
reservation \n")
           self.started= True
       else:
           self.available_seats.empty_heap()
           self.waitlist.empty_heap()
           self.red_black_tree = RBTree()
           self.add_seats(seat_count)
           self.output_file.write(f"{seat_count} Seats are made available for 
reservation \n")
```

Note: whenever an initialize is used second time in between a set of commands the entire system will be re initialized from beginning including the reservations, waitlist and the available seats

2)*Available()*: available will simple print out the number of seats not reserved(i.e length of the available seats min heap)

```
def available(self):
       available_seat_count = len(self.available_seats.available.heap)
       waitlist_length = len(self.waitlist.waitlist.heap)
       self.output_file.write(f"Total Seats Available : {available_seat_count}, 
Waitlist : {waitlist_length} \n")
```

3)*Reserve(userID, userPriority):* if there is already an entry with the given userID, will return the same and exit. Else checks whether seats are available, if seats are available then checks whether waitlist is empty or not, if not empty, adds the (userid, userpriority) to the waitlist and pops the waitlist priority queue(in order to get the highest priority for reservation) and adds the popped user from waitlist with popped seat_id from available seats to the redblack tree. if waitlist is empty, directly adds the user_id, seat_id to redblack tree.

```
def reserve(self, user_id, user_priority):
       seat_id = self.available_seats.get_seat()
       if self.red_black_tree.search_tree_helper(user_id) != None:
           self.output_file.write(f"User {user_id} is already has a reservation with
{self.red_black_tree.search_tree_helper(user_id).seat_id} \n")
           return
       if seat_id is not None:
           if len(self.waitlist.waitlist.heap) == 0:
               self.red_black_tree.insert_node(user_id, seat_id)
               self.output_file.write(f"User {user_id} reserved seat {seat_id} \n")
           else:
               temp = [user_id, user_priority]
               self.waitlist.enqueue(temp)
               first_priority = self.waitlist.dequeue()
               self.red_black_tree.insert_node(first_priority[0], first_priority[1])
               self.output_file.write(f"User {user_id} is added to the waiting list")
               self.output_file.write(f"User {first_priority[0]} reserved seat
{first_priority[1]} \n")
       else:
           self.waitlist.enqueue([user_id,user_priority])
           self.output_file.write(f"User {user_id} is added to the waiting list \n")
```

4)*Cancel(seat_id, user_id):* checks for the seat id, user_id pair in the redblack tree, if found deletes the pair and pops the waitlist and assigns the deleted seat_id with the popped user id. If the waitlist is empty, the the deleted seat id is simply added back in to the available seats min heap using re_adding function.

```
def cancel(self, seat_id, user_id):
       if self.red_black_tree.delete_node(user_id,seat_id):
           self.available_seats.re_adding(seat_id)
           self.output_file.write(f"cancelled from RB tree {user_id} and seat_id
{seat_id} \n")
           if len(self.waitlist.waitlist.heap) >0:
               seat_id = self.available_seats.get_seat()
               first_priority = self.waitlist.dequeue()
               self.red_black_tree.insert_node(first_priority[0], seat_id)
```

```
                self.output_file.write(f"User {first_priority[0]} reserved seat
{seat_id} \n")
        else:
            # search in waitlist
            if self.waitlist.remove_multi(user_id):
                self.output_file.write("removed from waitlist \n")
                self.available_seats.re_adding(seat_id)
```

5)*ExitWaitlist(userID):* searches the waitlist for the userID if found deletes the entry.

```
def exit_waitlist(self, user_id):
        if self.waitlist.remove_multi(user_id):
            self.output_file.write("deleted from waitlist \n")
            return
        self.output_file.write(f"user {user_id} not found in waitlist and reservation
list \n")
        return
```

6)*UpdatePriority(userID,userPriority):* checks for the userid in the waitlist, if found the userPriority in that entry is updated.

```
def update_priority(self, user_id, user_priority):
        if self.waitlist.update_priority(user_id,user_priority):
            self.output_file.write(f"User {user_id} priority has been updated to
{user_priority} \n")
        else:
            self.output_file.write(f"User {user_id} priority is not updated  \n")
```

Note: after any entry/delete the waitlist/available seats(basically any data structure using the min-heap) will heapify. Now for waitlist we are also tracking the order of entry of a userid (simoilar to timestamp) with a variable called counter. When the update function is invoked we will find the entry with the userid, store the value of the counter in temp variable, delete the entry from waitlist and re-add the (userid, updated userPriority, counter = temp) into the waitlist.

7)*AddSeats(count)*: add count number of seats starting from the largest existing seat id. I.e(largest_seat_id to largest_seat_id +count).if some entries are present in the waitlist after addition they will be added in to reservation such that either waitlist or available seats will be empty.

```
def add_seats(self, count):
        self.available_seats.adding(count)
        self.output_file.write(f"Additional {count} Seats are made available for
reservation \n")
```

```
        loop_count =
min(len(self.available_seats.available.heap),len(self.waitlist.waitlist.heap))
        print(f"min value is {loop_count} \n")
        for i in range (0,loop_count):
            seat_id = self.available_seats.get_seat()
            user_id = self.waitlist.dequeue()[0]
            self.red_black_tree.insert_node(user_id,seat_id)
            self.output_file.write(f"User {user_id} reserved seat {seat_id} \n")
```

8)*PrintReservations()*: the redblack tree inorder traversal will be printed out for each node making it in the ascending order of seat id, as the red-black tree is built considering the seat_id comparisons.

```
def print_reservations(self):
        self.output_file.write(self.red_black_tree.inorder_traversal())
        pass
```

9)ReleaseSeats(userID1,userID2): deletes the userids ranging from userid1 to userid2, from both redblack tree and the waitlist, after deletion fills back the released seats from userids from the waitlist. In delete_node function i have added another parameter cancel==True, to make note of where the delete call is happening.as for release seats irrespective of the seat_ids of user, the nodes of user ids have to be removed, but for cancel, we need to consider both userid and seatid before deleting from reservation.

```
def release_seats(self, user_id1, user_id2):
        for user_id in range(user_id1,user_id2+1):
            if self.red_black_tree.search_tree_helper(user_id)!= None:
                seat_id = self.red_black_tree.search_tree_helper(user_id).seat_id
                self.red_black_tree.delete_node(user_id,seat_id=None, cancel=False)
                self.available_seats.re_adding(seat_id)
            else:
                self.waitlist.remove_multi(user_id)
        loop_count =
min(len(self.available_seats.available.heap),len(self.waitlist.waitlist.heap))
        self.output_file.write("deleted from waitlist \n")
        for i in range (0,loop_count):
            seat_id = self.available_seats.get_seat()
            user_id = self.waitlist.dequeue()[0]
            self.red_black_tree.insert_node(user_id,seat_id)
            self.output_file.write(f"User {user_id} reserved seat {seat_id} \n")
```

10)Quit(): exits the program at this point.

```
def quit(self):
```

```
        self.output_file.write("Program Terminated!! \n")
        sys.exit()
```

Implementation of data structures:

RED-BLACK TREE:

- I have implemented the red black tree for maintaining the reservations because that is a self balancing tree binary search tree, which decreases the time to search a value(O(log n)).
- The seat_id is considered for sorting and building the tree.
- The tree self balances after every deletion, insertion.
- Contains helper functions for satisfying the requirements mentioned in the project.

MIN-HEAP:

- The choice to use min-heap for maintaining available seats is because of the requirement to allocate the least possible seat_id to a given user and its just a simple min heap with single value entries..
- Ideally an additional max-heap data structure should have been used for maintaining the waitlist because of the highest priority preference, in order to make the min-heap work for the waitlist as well, we have used negating(-1*user_priority) the value of user_priority during insertion into the waitlist.
- Implemented few additional functions only to work with either waitlist/available seats, and few functions common for both.
- For insertion in to waitlist, firstly the input parameter will be a list ([user_id,seat_id],counter= None) we will be adding one more value counter = counter+1 (declared 0 in __init__) to the entry making it [user_id, seat_id, counter] if counter is None, else we will directly pass the parameter value of counter as the 3rd element of entry.
- Heapify_insert and heapify_delete functions will make sure that the property of min-heap is maintained after any operation.
- Heapify_insert_single/heapify_delete_single for available seats where the value is used for comparison.
- heapify_insert_multi/heapify_delete_multi for waitlist, where the user_priority value among the keys has to be used for comparison.
- In case of priority updates/any operations making the priority value of two elements same, then the lowest counter value is considered.
- Even though the update priority implemented works fine, As min heap does not provide the feasibility priority updates directly, there needs to be 2 extra steps in the form of deletion and re-insertion of the element which can effect efficiency in cases where priorities need to be updated more frequently.
- Reusing the same code for min-heap for available seats in the waitlist priority queue, reducing the code redundancy.

The priority-queue uses min-heap, enqueue will be inserting into the min heap, deque for popping the heap, and update_priority to take care of updations without changing the counter value associated with the userid.

<u>Make commands</u>

1)Build: `make build`

- Creates virtual environment, installs dependencies and then companies the executable
- Create Virtual Environment: Sets up a virtual environment named env.
- Activate Virtual Environment: Activates the virtual environment for dependency isolation.
- Install Dependencies: Installs the packages listed in requirement.txt.
- Compile Executable: Uses PyInstaller to create a standalone executable from main.py, outputting it to the dist directory.

2)Run directly(without any executable: `make run input=input_file.txt`

- Runs the python code with the input file. *does not create any executable file.uses typica python3 file.py execution.

3)Run Executable: `make run_executable input=input_file.txt`

- Runs the standalone executable file with the input

4)Clean: `make clean`

- Deletes generated files and directories, restoring the project to a clean state

5)Help: `make help`

- Displays a summary of all the available targets and usage instructions

6)All: `make all`

- Provides basic information about the makefie and usage guidance

Note:

- python usually does not create an executable file to run, in order to do that i have used a python package pyinstaller, which allows to create an executable but that package needs to be installed first, hence created another requirements.txt file containing the package name which is being run to install in build target.
- Using the pyinstaller package results in creation of an executable file along with a couple of dependency files/folders, the executable file will be generated in the folder called Dist.
  I.e path to executable = curr_directory/dist
- __init__.py file needs to be maintained in order for the imports to work.
- The make run_executable command is not capable of executing in windows due to no available support for pyenv. But is compatible with macOS as tested in multiple devices.

References:

https://www.geeksforgeeks.org/introduction-to-red-black-tree/
https://www.geeksforgeeks.org/introduction-to-min-heap-data-structure/
https://makefiletutorial.com/
https://docs.python.org/3/library/re.html