# The Gator Ticket Master

**Name**: Deepika Muchukota

**UFID**: 55474390

**UF Email**: dmuchukota@ufl.edu

## Structure of the Program

The program consists of three main components:

- RedBlackTree class (for efficient seat reservation management)
- MaxHeap class (for priority-based waitlist management)
- GatorTicketMaster class (main system controller)

**RedBlackTree Implementation**

The RedBlackTree class provides O(log n) operations for managing seat reservations where:

- Nodes store (user_id, seat_id) pairs
- User_id serves as the key for searching
- Color property maintains tree balance
- NIL sentinel nodes handle boundary cases

Key features of RedBlackTree:

- Self-balancing binary search tree
- Maintains O(log n) height through color properties
- Efficient search, insert, and delete operations
- In-order traversal for sorted seat information

The tree structure provides methods for:

- Inserting new reservations
- Finding user reservations
- Deleting reservations
- Maintaining tree balance through rotations

**MaxHeap Implementation**

The MaxHeap class manages the priority waitlist where:

- Elements are stored as tuples (priority, user_id, timestamp)

- Higher priority numbers get precedence
- Earlier timestamps break priority ties
- Array-based implementation for efficiency

**Key features of MaxHeap:**

- Complete binary tree representation
- Parent's priority ≥ children's priorities
- O(log n) operations for all modifications
- Timestamp-based ordering for equal priorities

**GatorTicketMaster Implementation**

The GatorTicketMaster class coordinates the entire system. It manages:

- Available seats using a list
- Reserved seats using RedBlackTree
- Waitlist using MaxHeap

Key components:

1. Seat Management:
    - Tracks available and reserved seats
    - Assigns lowest numbered available seat
    - Uses RedBlackTree for O(log n) operations
    - Handles seat cancellations and releases
2. Waitlist Management:
    - Priority-based ordering of waiting users
    - Automatic seat assignment when available
    - Priority updates and user removal
    - Maintains timestamp ordering
3. Data Structures Used:
    - RedBlackTree (reserved_seats): Manages seat reservations
    - List (available_seats): Tracks available seat numbers
    - MaxHeap (waitlist): Manages priority-based waiting list

The system ensures:

- O(log n) reservation operations
- Priority-based waitlist management

- Proper handling of cancellations
- Dynamic seat addition capability
- Efficient range-based seat releases

**Program Flow:**

Program Flow

The program execution follows a systematic flow for managing seat reservations and waitlist operations:

Command Processing

The program begins by reading commands from an input file. Each command is parsed to extract the operation type and its arguments. These commands are then executed by the GatorTicketMaster class, which coordinates between the RedBlackTree (for seat reservations) and MaxHeap (for waitlist management).

Seat Management

When processing seat reservations:

- The system checks seat availability in the available_seats list
- If seats are available:
    - Assigns the lowest numbered available seat
    - Uses RedBlackTree to store the reservation with O(log n) efficiency
    - User_id serves as the key and seat_id as the value
- If no seats are available:
    - User is added to MaxHeap-based waitlist
    - Priority and timestamp determine position in waitlist
    - Higher priority numbers get precedence
    - Earlier timestamps break priority ties

Cancellation Handling

For seat cancellations:

- System verifies reservation using RedBlackTree's find operation
- If reservation exists:
    - Removes user from RedBlackTree using delete operation
    - If waitlist exists:
        - Assigns seat to highest priority user from MaxHeap
        - Updates RedBlackTree with new reservation

        - If no waitlist:
            - Adds seat back to available_seats list

Dynamic Seat Addition

When new seats are added:

- System calculates next available seat number using RedBlackTree traversal
- For each new seat:
    - If waitlist exists:
        - Assigns seats to users based on MaxHeap priority order
        - Updates RedBlackTree with new reservations
    - Remaining seats added to available_seats list

Range-based Operations

For releasing seats in a user ID range:

- Uses RedBlackTree's in-order traversal to find affected reservations
- Removes reservations from RedBlackTree
- Removes users from MaxHeap waitlist within range
- Reassigns released seats to waitlisted users based on priority

Output Generation

- Each operation generates appropriate output messages
- Results are written to output file with same name as input file
- Maintains proper error handling and status reporting

This implementation ensures:

- O(log n) efficiency for reservation operations using RedBlackTree
- Priority-based waitlist management using MaxHeap
- Proper balance between tree nodes through Red-Black properties
- Consistent seat numbering and assignment
- Efficient range-based operations
- Reliable error handling and output generation

The combination of RedBlackTree and MaxHeap provides an efficient and robust system for managing seat reservations while maintaining proper ordering and priority handling.

## Function Prototypes:

## In Red-Black tree:

```
left_rotate(self, x):
    # Performs left rotation around node x to maintain tree balance and red-black properties

right_rotate(self, x):
    # Performs right rotation around node x to maintain tree balance and red-black properties

insert(self, user_id, seat_id):
    # Inserts new node with user_id as key and seat_id as value, maintaining red-black properties

insert_fixup(self, k):
    # Fixes red-black tree violations after insertion through rotations and color changes

find(self, user_id):
    # Finds and returns node with given user_id, or None if not found

delete(self, user_id):
    # Deletes node with given user_id and maintains red-black properties

delete_fixup(self, x):
    # Fixes red-black tree violations after deletion through rotations and color changes

transplant(self, u, v):
    # Helper method for delete operation to replace subtree u with subtree v

in_order_traversal(self):
    # Returns sorted list of (seat_id, user_id) pairs through in-order traversal
```

## IN Max_heap:

```
parent(self, i: int) -> int:
    # Returns parent index using formula (i-1)//2

left_child(self, i: int) -> int:
    # Returns left child index using formula 2i + 1

right_child(self, i: int) -> int:
    # Returns right child index using formula 2i + 2

_compare(self, a: tuple, b: tuple) -> int:
    # Compares tuples by priority first, then timestamp for equal priorities

insert(self, key: tuple):
    # Adds new (priority, user_id, timestamp) tuple and maintains max heap property

pop(self) -> tuple:
    # Removes and returns highest priority tuple (priority, user_id, timestamp)

remove(self, user_id: int) -> bool:
    # Removes specific user from waitlist, returns success status

update_priority(self, user_id: int, new_priority: int) -> bool:
    # Updates user's priority in waitlist, returns success status

contains(self, user_id: int) -> bool:
    # Checks if user exists in waitlist
```

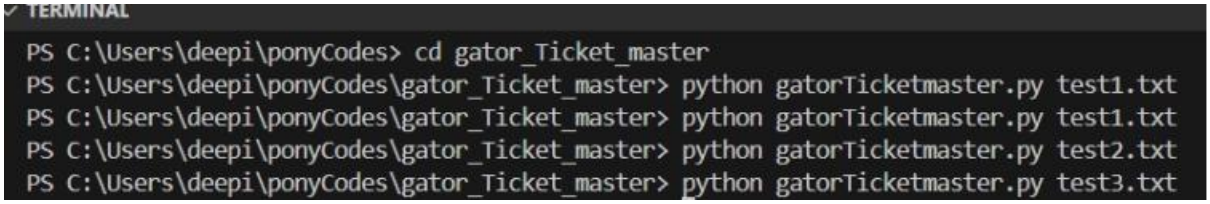
## In gatorTicketMaster:

```
initialize(self, seat_count: int):
    # Initializes system with given number of seats

available(self):
    # Returns current count of available seats and waitlist size

reserve(self, user_id: int, user_priority: int):
    # Handles seat reservation or adds to waitlist if no seats available

cancel(self, seat_id: int, user_id: int):
    # Cancels reservation and reassigns seat to highest priority waitlisted user

exit_waitlist(self, user_id: int):
    # Removes user from waitlist if present

update_priority(self, user_id: int, new_priority: int):
    # Updates priority of waitlisted user

add_seats(self, count: int):
    # Adds new seats and assigns to waitlisted users by priority

release_seats(self, user_id1: int, user_id2: int):
    # Releases seats in user ID range and reassigns to waitlisted users

print_reservations(self):
    # Returns all current reservations sorted by seat number

quit(self):
```

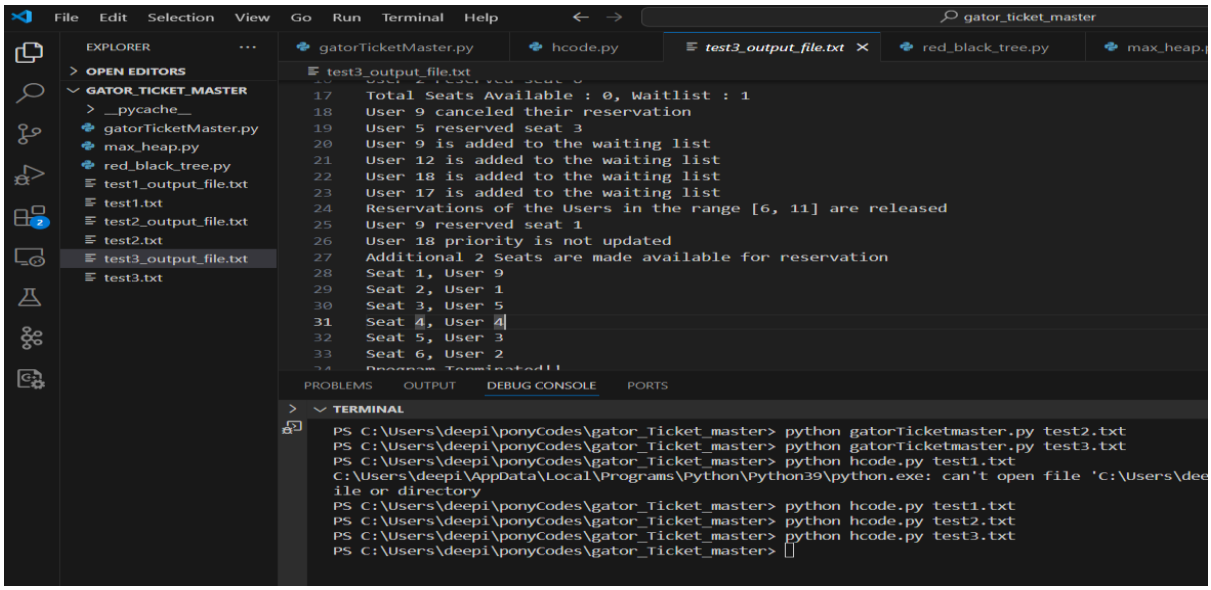
# Terminates program execution

**Execution:**

1.Create the input file (test1.txt) and it includes the methods that should be invoked in the program.
2.From the terminal just run the command to execute the code




3.When it successfully executes  an output file named test1_output_file.txt is generated within the directory .




**<u>No Makefile Provided:</u>**

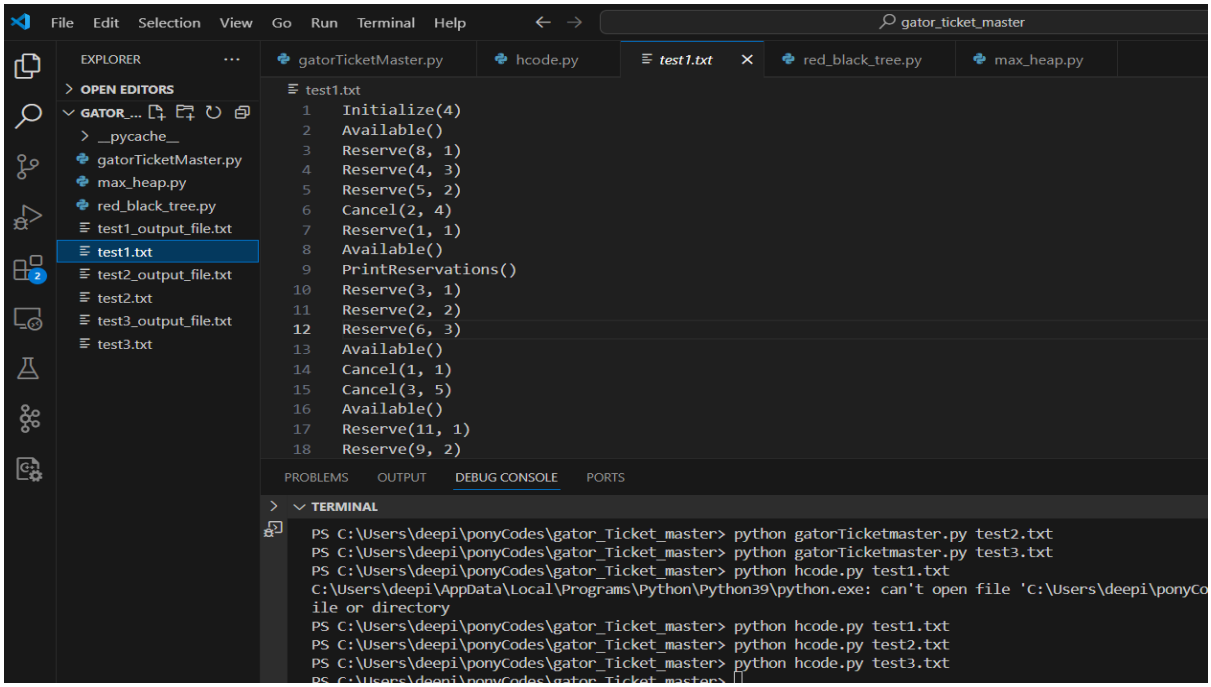
This project does not include a makefile, as it is implemented in Python. Since Python is an interpreted language, it does not require a makefile—the source code is executed directly by the Python interpreter without the need for separate compilation.

**Conclusion:**

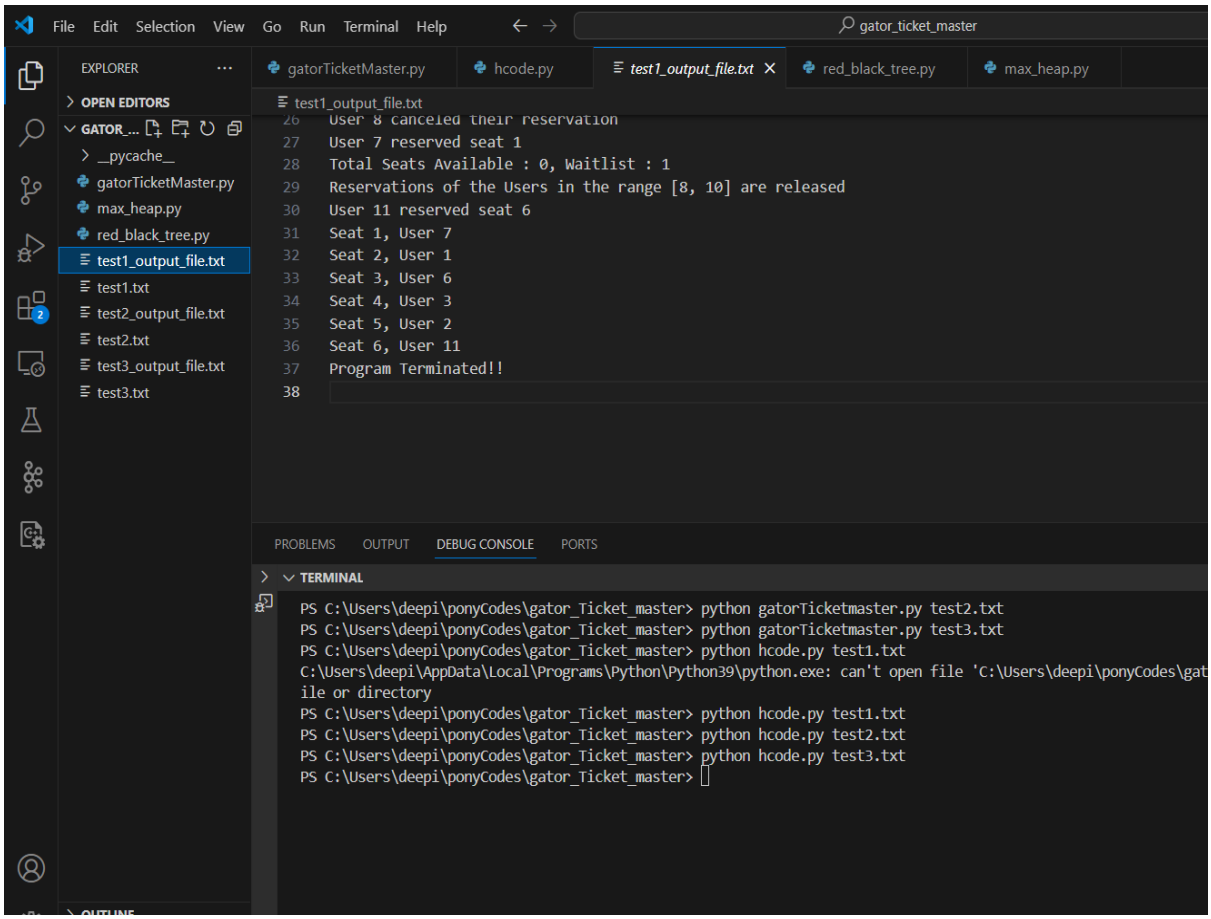
The program was tested using the inputs from the *TestCases_v2.pdf* file, and it was observed that the program performed as expected, with the output matching the anticipated results.

# Final Results:

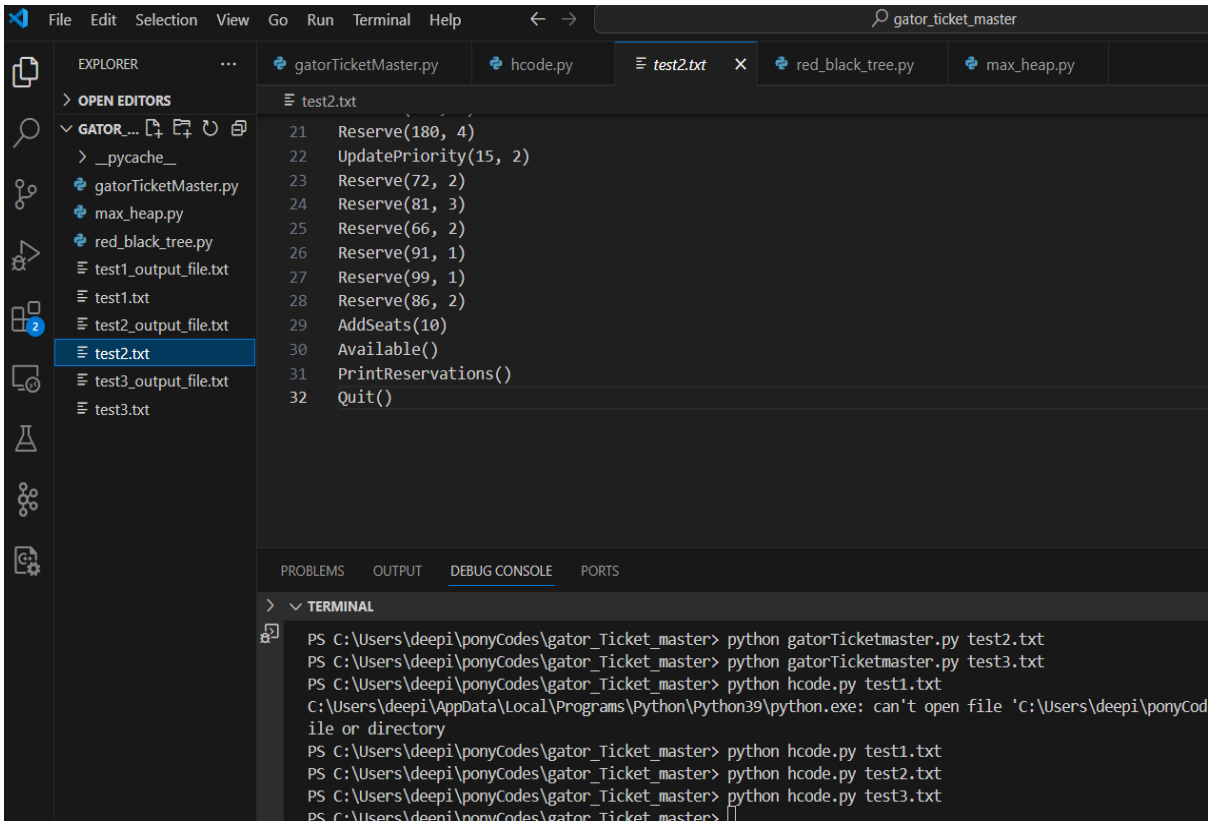
## Test Case1:



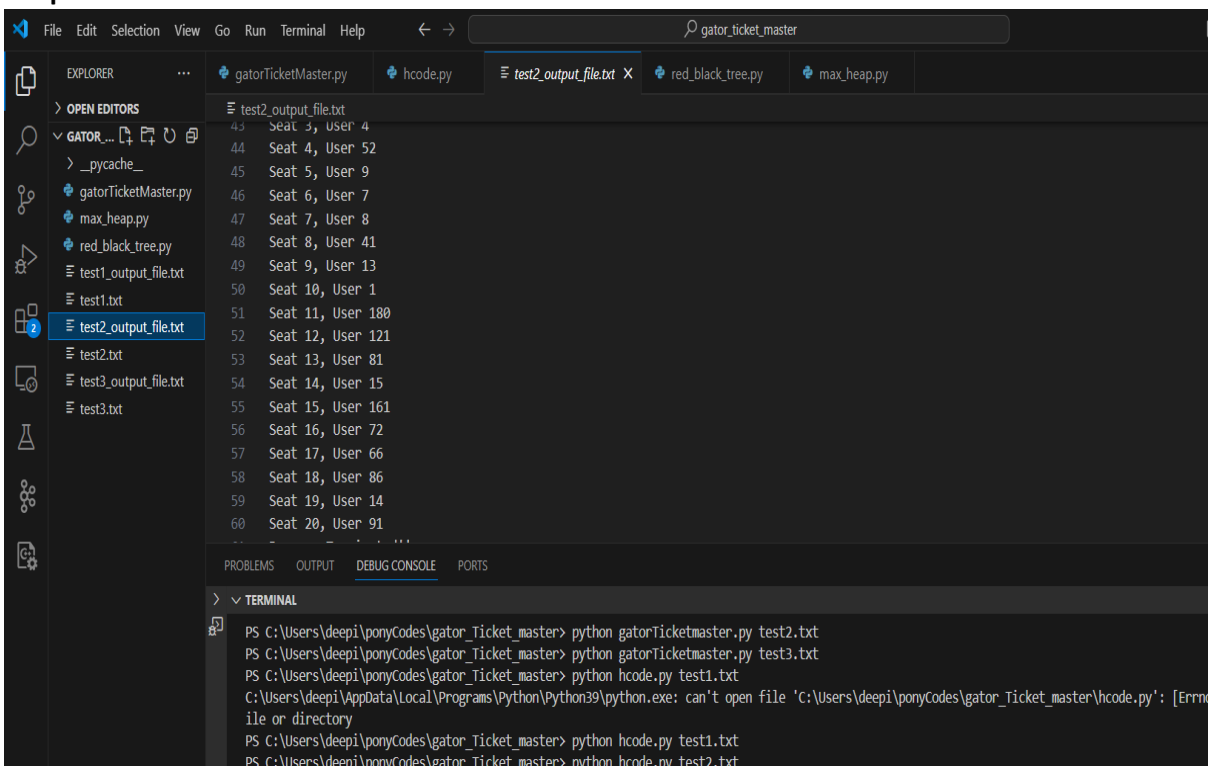

## Output1:



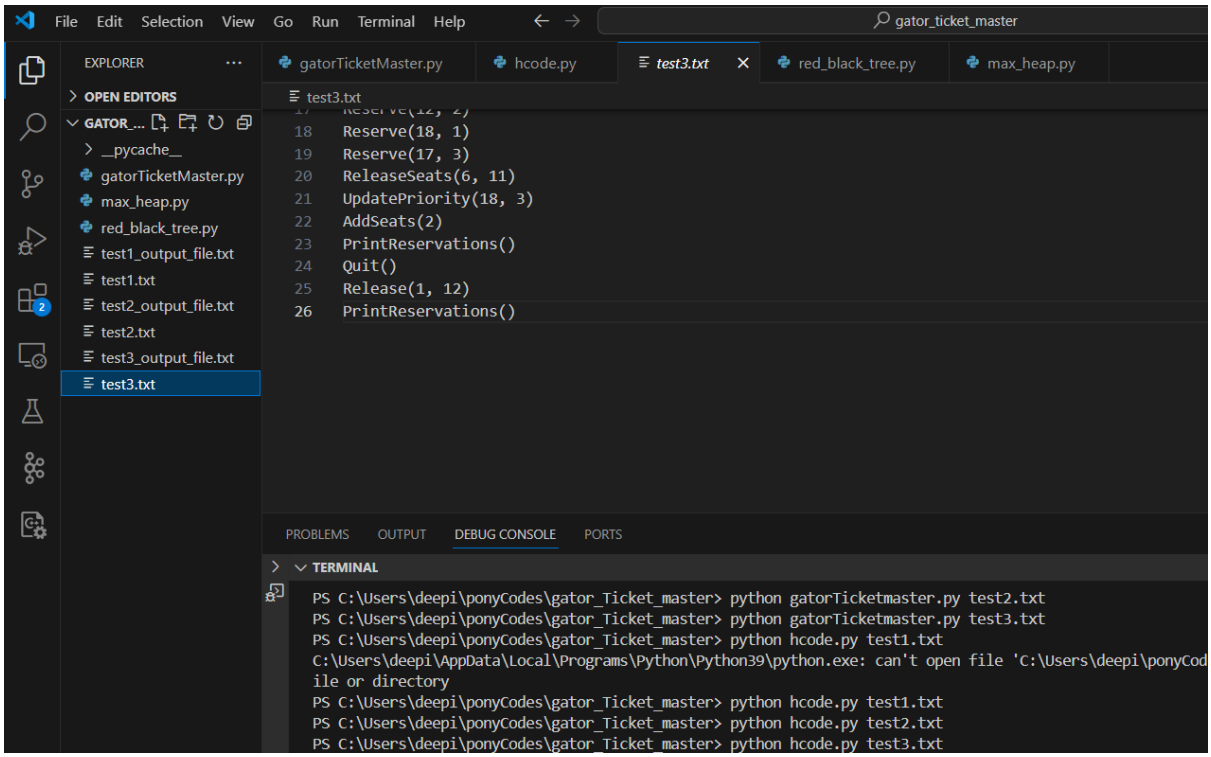
## Test Case2:




## Output2:



## Test Case3:




## Output3: