Gator Ticket Master

Date: 11/14/2024

Name: Sanket Sunil Deshmukh
Email: sanket.deshmukh@ufl.edu
UFID: 32226339

How to execute the program?

1. Unzip the project from Deshmukh_Sanket.zip
2. On terminal, traverse inside the unzipped the folder
3. execute command "**make**", which will generate a file "**gatorTicketMaster**"
4. execute command "**python3 gatorTicketMaster fileName**" to run the program
5. verify/retrieve output stored in file "fileName_output_file.txt" placed at the same path

Problem Statement

Gator Ticket Master is a seat booking service for Gator Events. Users utilize this service to secure a seat for attendance at a gator event. The service is seeking to develop a software system that effectively manages seat allocation and reservation management.

Key Requirements

1. Seat Arrangement using RBT
2. Waiting List using Heap
3. List of available seats using Heap

Approach

1. Keeping the entire code in a single file would create a total mess and difficult to interpret the code in future.
2. Hence, I have built different packages and interacting with them through single file **GatorTickerMaster.py**
3. For **seatMapping** we were asked to use **Red Black Tree (RBT)**, the code for operations can be found in **rbt.py**
    a. **Each node** in RBT consists of : **userId**, **seatId**, **color** (node color = black/red), **left** (left child pointer), **right** (right child pointer), **parent** (parent node pointer)
    b. **Tree Structure:** Initialize the Tree with External Node as root Node (as initially the tree is empty)

   c. **Note:** I am also using external node in RBT, which is denoted as **EXT_NODE** node, which has **userId = seatId = None** (As it is an external node)
4. For managing the **waitlist**, we were asked to use heap. Since, 2 > 1 for priorities. I decided to use **Max Heap** (**binMaxHeap.py**) here, it will allow us to bring in max priority at the top of the heap.
   a. I have used an array to store the heap (as it helps in easy calculations of parent and child node pointers.
   b. Each index in array corresponds to node in Binary Max Heap
   c. **At each index**:
      i. **priority**: Used to store the user priority
      ii. **timestamp**: Used to store the timestamp (in nanoseconds) at which the user entered the wait list.
      iii. **userId**: Unique Identification of the user standing at that position in the waitlist heap
5. For managing the **availability of seats**, we were asked to use **Min heap** (**binMinHeap.py**)
   a. Here also, I am using an array to store the min heap for ease of calculations.
   b. At each index, I am mapping the seat number which is available

Function Prototypes and explanations (for each class):

 # class to implement the functions for processing each user command
GatorTicketMaster:

1. __init__(self):
   # Constructor to initialize variables for data structures

2. initialize(self, n):
   # Initialize seats with n number of seats

3. available(self):
   # Check and display available seats and waitlist

4. reserve(self, userId, priority):
   # Reserve a seat for a user with given priority

5. cancel(self, userId, seatId):
   # Cancel a user's seat reservation

6. exitWaitList(self, userId):
   # Remove a user from the waiting list

7. addSeats(self, extraSeats):
   # Add additional seats to available seats

8. releaseSeats(self, userId1, userId2):
   # Release seats for users in a given range

9. printReservations(self):
   # Print all current seat reservations

10. updatePriority(self, userId, priority):
    # Update a user's priority in the waiting list

 # class to implement Node for Red Black Tree
Node (RBT):

1. def __init__(self, userId=None, seatId=None, color='red'):
   # Node constructor

class to implement Red Black Tree
RedBlackTree:

1. __init__(self):
   # RedBlackTree constructor

2. leftRotate(self, x):
   # Perform left rotation on node x

3. rightRotate(self, x):
   # Perform right rotation on node x

4. search(self, userId):
   # Search for a node with given userId
   # Returns: Node or EXT_NODE

5. findNode(self, node, userId):
   # Find a node with given userId starting from the given node
   # Returns: Node or EXT_NODE

6. swapNodes(self, delNode, childNode):
   # Handle pointer transfers during node deletion

7. minimum(self, node):

```
        # Find the minimum value node in the subtree
        # Returns: Node with minimum value
```

8. inorderTraversal(self, node):
```
        # Perform inorder traversal of the tree starting from the given node
        # Prints node details
```

9. insert(self, userId, seatId):
```
        # Insert a new node with given userId and seatId
```

10. insertNode(self, node):
```
        # Insert a node into the Red-Black Tree
```

11. fixInsert(self, node):
```
        # Fix Red-Black Tree properties after insertion
```

12. delete(self, userId):
```
        # Delete a node with given userId
```

13. deleteNode(self, node):
```
        # Delete a specific node from the Red-Black Tree
```

14. fixDelete(self, x):
```
        # Fix Red-Black Tree properties after deletion
```

class to implement Binary Min Heap

binMinHeap:

1. __init__(self, maxSize):
```
        # Constructor to initialize the heap with maximum size
```

2. isFull(self):
```
        # Checks if the heap is full
        # Returns: Boolean
```

3. numberOfAvailableSeats(self):
```
        # Returns the number of available seats
```

4. exchangeIndexValues(self, index1, index2):
```
        # Swaps values at two given indexes
```

5. calculateParentIndex(self, index):
```
        # Calculates and returns the parent index of a given index
```

6. calculateLeftChildIndex(self, index):
   # Calculates and returns the left child index of a given index

7. calculateRightChildIndex(self, index):
   # Calculates and returns the right child index of a given index

8. checkParent(self, index):
   # Checks if the given index has a parent
   # Returns: Boolean

9. checkLeftChild(self, index):
   # Checks if the given index has a left child
   # Returns: Boolean

10. checkRightChild(self, index):
    # Checks if the given index has a right child
    # Returns: Boolean

11. parent(self, index):
    # Returns the parent value of a given index

12. leftChild(self, index):
    # Returns the left child value of a given index

13. rightChild(self, index):
    # Returns the right child value of a given index

14. heapifyUp(self, index):
    # Performs heapify operation from bottom to top

15. heapifyDown(self, index):
    # Performs heapify operation from top to bottom

16. insert(self, data):
    # Inserts a new element into the heap

17. removeMin(self):
    # Removes and returns the minimum element from the heap

18. addSeats(self, extraSeats):
    # Adds extra seats to the heap

Class to implement Binary Max Heap
binMaxHeap:

1. __init__(self):
   # Constructor to initialize the heap
2. lengthofWaitlist(self):
   # Returns the current size of the waiting list

3. exchangeIndexValues(self, index1, index2):
   # Swaps values at two given indexes

4. calculateParentIndex(self, index):
   # Calculates and returns the parent index of a given index

5. calculateLeftChildIndex(self, index):
   # Calculates and returns the left child index of a given index

6. calculateRightChildIndex(self, index):
   # Calculates and returns the right child index of a given index

7. checkParent(self, index):
   # Checks if the given index has a parent
   # Returns: Boolean

8. checkLeftChild(self, index):
   # Checks if the given index has a left child
   # Returns: Boolean

9. checkRightChild(self, index):
   # Checks if the given index has a right child
   # Returns: Boolean

10. parent(self, index):
    # Returns the parent value of a given index

11. leftChild(self, index):
    # Returns the left child value of a given index

12. rightChild(self, index):
    # Returns the right child value of a given index

13. heapifyUp(self, index):

```
    # Performs heapify operation from bottom to top

14. heapifyDown(self, index):
    # Performs heapify operation from top to bottom

15. insert(self, userId, priority):
    # Inserts a new element into the heap

16. removeMax(self):
    # Removes and returns the maximum element from the heap

17. removeUser(self, userId):
    # Removes a specific user from the heap
    # Returns: Boolean indicating successful removal
```

Conclusion:

I have covered and verified all the required scenarios as per project description. I have also kept some extra functions like inorder_traversal() in a red black tree which can be used at any stage to check the state of the current seat mappings. More detailed description about the logic implemented for each function has been added via comments in the code.