DATA STRUCTURE AND ALGORITHMS

(Possible Questions)

General Questions

1. **Overview and Purpose**:
    - What is the main goal of this program?
    - Can you briefly explain the structure and flow of the program?

Answer:

Overview and Purpose:

- The program simulates a cinema management system, offering functionalities for viewing and managing movies by showtimes, booking seats, and prioritizing customer requests.
- **How it works**:
    - Movies are stored in a **min-heap**, which automatically sorts them by their showtimes.
    - Seats for each movie are managed using a **Binary Search Tree (BST)** for efficient addition, deletion, and traversal.
    - Priority customer requests are handled using a **max-heap**, ensuring customers with the highest priority (age) are processed first.

2. **Data Structures**:
    - Why did you use a **min-heap** for movies and a **max-heap** for priority customers?
    - What are the advantages of using a **binary search tree (BST)** for seat management instead of an array or a list?

Answer:

Data Structures:

- **Min-Heap (Movies)**: Ensures that movies are automatically ordered by their showtimes (earliest first). Internally, the heap uses a custom comparator to prioritize earlier showtimes.

- **Binary Search Tree (Seats)**: Organizes seats for efficient management (e.g., searching, booking, deleting) and displays them in order during traversal.
- **Max-Heap (Priority Requests)**: Uses the customer's age as a key to prioritize older customers. This is managed using a priority_queue where the highest age is always on top.

Specific Functionality Questions

1. **Movies**:
    - o How are movies sorted by showtime?
    - o What happens if two movies have the same showtime? How does the program handle that?

Answer:

Movies:

- o **How movies are sorted**:
  Each movie's showtime is converted to minutes from midnight (e.g., 1:00 PM becomes 13×60=78013 \times 60 = 78013×60=780 minutes). The min-heap stores these values, and the heap property ensures the movie with the smallest value (earliest time) is on top.
- o **Tie-breaking**:
  If two movies have the same showtime, the heap maintains the order of insertion.

2. **Seats**:
    - o Why is in-order traversal used for displaying seats?
    - o How does the program ensure that seat numbers remain unique?
    - o What is the difference between cancelBooking and deleteSeat?
    - o How is the deleteNode function implemented to handle different cases of deletion in the BST?

Answer:

1. **Seats**:

- o **How in-order traversal works**:
  In a BST, an in-order traversal visits the left subtree, processes the current node, and then visits the right subtree. This guarantees the seat numbers are displayed in ascending order.

- **Uniqueness of seat numbers**:
  The BST structure inherently prevents duplicate seats since insertion is based on comparison (<, >). Duplicate seat numbers will always follow the same path, ending up at a previously existing node.
- **Difference between cancelBooking and deleteSeat**:
  - cancelBooking: Marks a seat as available (isBooked = false) and then removes it from the tree.
  - deleteSeat: Directly removes the seat from the BST without checking its booking status.
- **How deleteNode works**:
  - It handles three cases:
    1. If the node has no children: It is directly deleted.
    2. If the node has one child: It is replaced by its child.
    3. If the node has two children: It finds the in-order successor (smallest node in the right subtree) and replaces the node's value with it before deleting the successor.

3. **Priority Queue**:
   - How does the program determine which customer gets priority in the queue?
   - What happens if two customers have the same age?

Answer:

Priority Queue:

- **How customers are prioritized**:
  The max-heap stores pairs (age, name) where the largest age determines the top priority.
  - When processing a request, the customer at the top of the heap is served, and the heap reorders itself to maintain the max-heap property.
- **Handling ties in age**:
  If two customers have the same age, the heap resolves ties based on insertion order (later entries come below earlier ones).

Code Design and Implementation Questions

1. **Efficiency**:
   - What is the time complexity of adding a seat, booking a seat, and viewing all seats?
   - What is the time complexity of managing the min-heap and max-heap operations?

Answer:

1. **Efficiency**:
   - **Time complexity**:
     - Adding a seat: O(log n)O(\log n)O(logn) (due to BST insertion).
     - Booking a seat: O(log n)O(\log n)O(logn) (BST search).
     - Viewing seats: O(n)O(n)O(n) (in-order traversal).
     - Heap operations: O(log n)O(\log n)O(logn) for insertion and deletion.
   - The program leverages efficient structures to minimize operational time.

2. **Error Handling**:
   - How does the program handle invalid inputs, such as booking a non-existent seat or selecting a movie that doesn't exist?
   - What happens if all seats are booked and a user tries to book another?

Answer:

1. **Error Handling**:
   - The program checks for invalid inputs or operations and displays appropriate error messages:
     - Booking a non-existent seat: Displays "Seat does not exist."
     - Booking an already booked seat: Displays "Seat is already booked."
     - Choosing an invalid movie: Displays "Invalid movie selection."
   - These checks prevent runtime errors and ensure proper user interaction.

3. **Code Clarity and Modularity**:
   - Why did you choose to use a map for associating movies with their seats?
   - Could this program be easily extended to support more features (e.g., dynamic movie addition)?

Answer:

1. **Code Modularity**:
   - The program is organized into classes for specific tasks:
     - Movie: Represents a movie with its name and showtime.
     - SeatBST: Manages seat operations for each movie.
     - PriorityQueue: Handles customer requests.
   - This separation ensures clarity, reusability, and ease of maintenance.

User Interaction Questions

1. **Menu Options**:
   - What happens if the user enters an invalid menu choice?
   - Is it possible to switch between movies after selecting one? If so, how?
2. **Usability**:
   - How does the program ensure a smooth user experience (e.g., feedback messages, input validation)?
   - Can a user cancel a booking without deleting the seat? How?

Real-Life Application and Challenges

1. **Practical Considerations**:
   - How could this system handle a larger scale, like a multiplex cinema with multiple screens?
   - What modifications would you make to adapt the program for online ticket booking?

Answer:

1. **Practical Considerations**:
   - **Large scale**: Extend the map to associate movies with screen IDs, allowing the addition of multiple screens.
   - **Dynamic movie addition**: Allow movies to be added during runtime by pushing new entries into the min-heap.

   Menu Options:
   - The menu guides the user through available options, with descriptive prompts and error messages for invalid inputs.

- Switching movies is seamless by selecting a new movie in the "Choose Movie" option, resetting the current context.

2. **Potential Issues**:
    - What are the limitations of using a BST for seat management? How could this be improved?
    - What happens if there's a power outage or the program crashes? How would you ensure data persistence?

Answer:

1. **Potential Issues**:
    - **Unbalanced BST**: As the BST grows, it may become unbalanced, reducing efficiency. A self-balancing BST (like AVL or Red-Black Tree) would ensure optimal performance.
    - **Data persistence**: Current data is lost when the program ends. Adding file I/O can save and reload seat and movie data.

Advanced Questions

1. **Algorithm and Logic**:
    - Can you explain the logic behind the deleteNode function in the BST? Why do you find the **in-order successor**?
    - Why did you use a custom comparator in the Movie struct to achieve a min-heap?

Answer:

1. **Algorithm and Logic**:
    - The min-heap orders movies using a comparator that ensures the smallest showtime is at the root.
    - The BST's recursive nature ensures that operations like insertion, deletion, and traversal remain efficient and intuitive.

2. **Extensibility**:
    - How would you implement a feature to allow seat reassignment?
    - Could this system support dynamic pricing for seats? If so, how would you implement it?

Answer:

Extensibility:

- Features like dynamic pricing can be added by including pricing attributes in the SeatNode class. The program could adjust prices based on demand and time to showtime.

3. **Comparison**:
   - If you had to replace the BST with another data structure, which one would you choose and why?

Answer:

Comparison:

- A hash table could replace the BST for faster seat management, but it would lose the ability to display seats in order. Thus, a BST is more suited for the current requirements.

Team and Presentation Questions

1. **Team Collaboration**:
   - How did you divide the tasks among team members while working on this project?
   - What were the biggest challenges your team faced, and how did you overcome them?

Answer:

Team Collaboration:

- We divided the work into modules (movies, seats, and customer priority). Each member handled specific functionality, and we integrated the code via a shared repository.
- Testing was performed collaboratively to resolve integration issues.

2. **Learnings**:
   - What did you learn from implementing this system?
   - If given more time, what changes or improvements would you make to the program?

Answer:

1. **Learnings**:
   - We learned how to apply data structures like heaps and BSTs to real-world scenarios and structure modular, maintainable code.
   - Given more time, we would implement persistent data storage, dynamic pricing, and support for multiple cinema screens