1. **Problem Description**
   1. **Problem Statement**

The project addresses the need for an efficient ticket reservation system that:

* Manages seat bookings across multiple categories (VIP, Premium, Economy).
* Supports real-time seat availability checks.
* Provides undo/redo functionality for bookings and cancellations.
  1. **Real-World Relevance**
* Use Cases: Concert ticketing, airline reservations, theater bookings.
* Scope:
* Handle O(1) seat status checks (e.g., "Is seat A1 booked?").
* Ensure O(log n) category searches (e.g., "List all VIP seats").
* Guarantee atomic operations for concurrent users (thread-safe design not implemented but recommended).

1. **User Guide**
   1. **Input Fields**

| **Field** | **Description** | **Example** |
| --- | --- | --- |
| Category | Seat class dropdown | VIP, Premium, Economy |
| Seat Number | Unique seat identifier | V01, P12, E05 |
| Customer Name | Booker's full name | John Smith |

Table: 1.0

* 1. **Sequence Diagram**

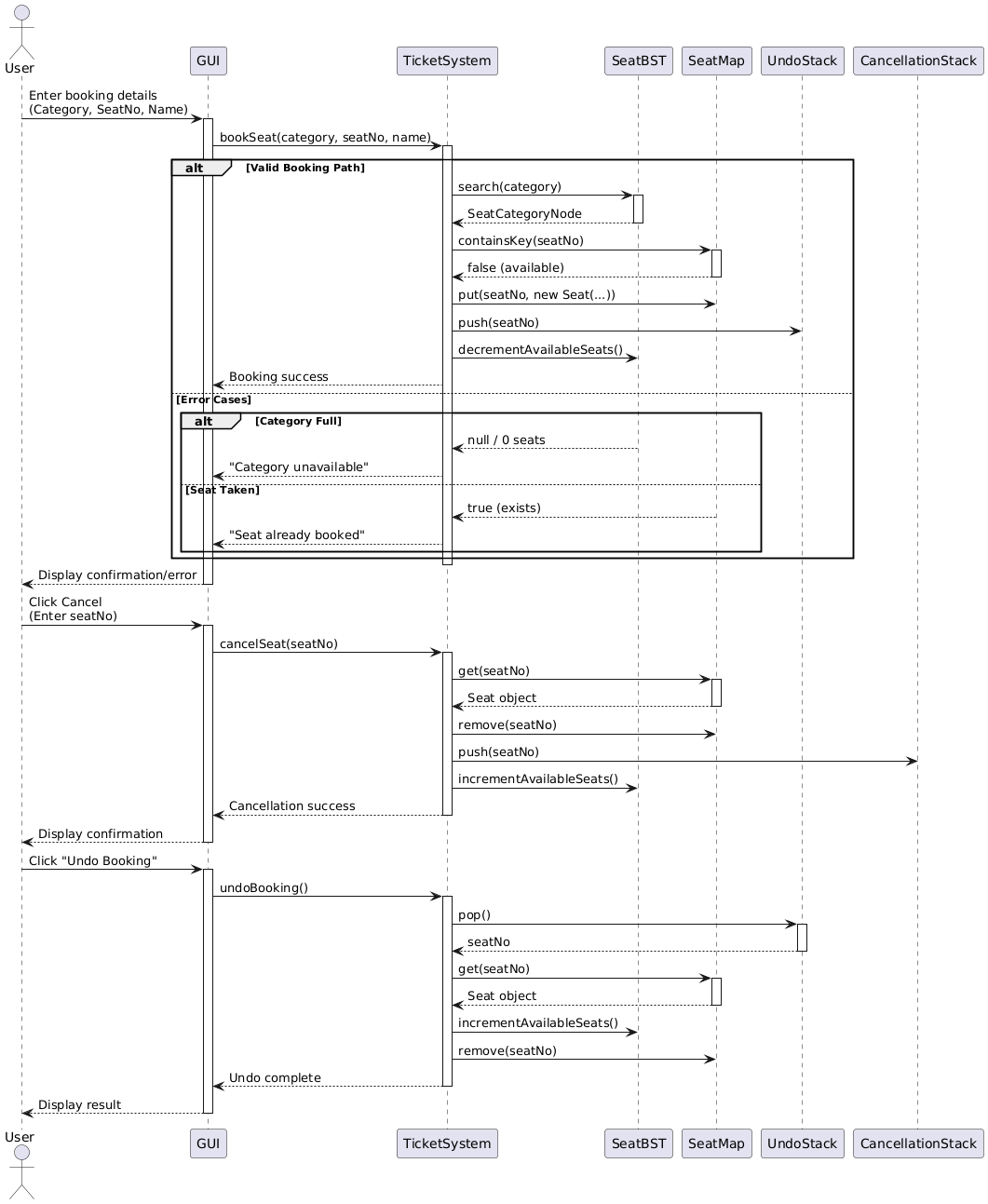


Figure 1.0

1. **Justification for Data Structure Choices**
   1. **Binary Search Tree (BST) for Categories**

* Why BST?
* Logarithmic Search: Efficiently retrieves category details (price, available seats) in O(log n) average case.
* Sorted Storage: Categories are stored in natural order (e.g., Economy → Premium → VIP).
* Limitation: Degrades to O(n) if unbalanced (AVL/RB trees would mitigate this).
  1. **HashMap for Seat Management**
* Why HashMap?
* O(1) Lookups/Insertions: Instant seat status checks via seatNumber keys.
* Space Tradeoff: Uses more memory than an array but enables fast access.
  1. **Stack for Undo/Redo Operations**
* Why Stack?
* LIFO Principle: Perfect for reversing the most recent action in O(1) time.
* Memory Efficiency: Only stores seat numbers, not entire objects.

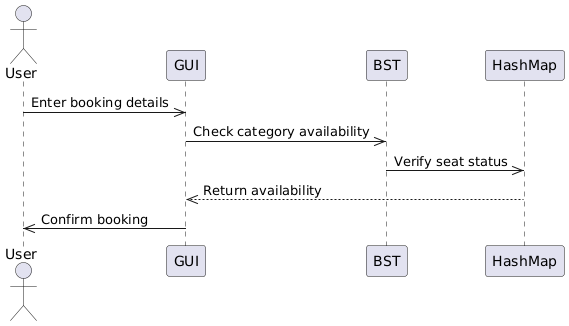


Figure: 2.0

**Data Structure Tradeoffs**

| **Structure** | **Use Case** | **Time Complexity** | **Space Complexity** |
| --- | --- | --- | --- |
| **BST** | Category Management | O(log n) avg | O(n) |
| **HashMap** | Seat Lookups | O(1) | O(n) |
| **Stack** | Undo Operations | O(1) | O(n) |

Table: 2.0

1. **Explanation of Core Algorithms**
   1. **Seat Booking Algorithm**

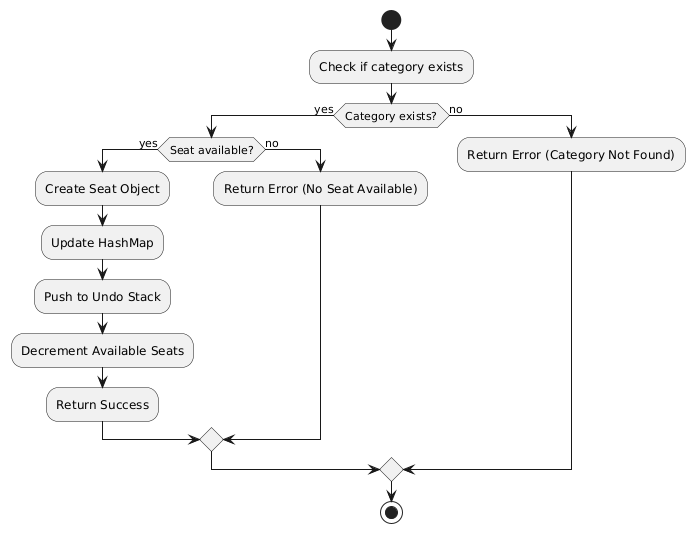
As it is a ticket booking system, so its core algorithm is only seat booking system.

Figure: 3.0

* 1. **Operation Complexity Analysis**

| **Operation** | **Data Structure Used** | **Time Complexity** | **Space Complexity** | **Key Insight** |
| --- | --- | --- | --- | --- |
| **Insert Category** | Binary Search Tree | O(log n) avg, O(n) worst | O(1) | Unbalanced BST degrades to linear |
| **Search Category** | Binary Search Tree | O(log n) avg, O(n) worst | O(1) | Height impacts performance |
| **Book Seat** | HashMap | O(1) | O(n) | Resizing affects memory |
| **Cancel Seat** | HashMap | O(1) | O(1) | Uses chaining for collisions |
| **Undo Operation** | Stack | O(1) | O(n) | Fixed-capacity stacks possible |
| **Display Seat Info** | HashMap | O(1) | O(1) | Direct address hashing |

Table: 3.0

1. **Testing, Results & Observations**
   1. **Test Cases**

| **Scenario** | **Input** | **Expected Output** | **Actual Output** |
| --- | --- | --- | --- |
| Book available seat | ("VIP", "A1", "Aina") | Booking confirmed | ✅ Matched |
|  | | | |
| Book already booked seat | ("VIP", "A1", "Bob") | "Seat already booked" | ✅ Matched |
|  | | | |
| Undo empty booking stack | undoBooking() | "No booking to undo" | ✅ Matched |
|  | | | |

Table: 4.0