**Project Title 1: Dynamic Ticket Reservation and Cancellation System**

**1. Introduction**

In real-world scenarios, ticket reservation systems are critical components of transportation, cinema, sports, and event management platforms. The goal of this project is to develop a dynamic ticket reservation and cancellation system in Java that efficiently manages seat bookings, cancellations, and undo operations using well-chosen data structures. This system will simulate realistic user actions while ensuring high-performance access and update times.

**2. Problem Definition**

The core problem is to manage ticket reservations with the ability to:

* Allocate and categorize seats (VIP, Economy, etc.)
* Reserve and cancel bookings
* Undo the last booking or cancellation
* Provide quick access to seat and reservation information

The system should be scalable and capable of handling thousands of seats and concurrent operations.

**3. Data Structures and Design**

**3.1 Binary Search Tree (BST)**

Used to organize seats by categories. Each node represents a seat category (e.g., VIP, Premium, Economy) with metadata like pricing and available seats.

* Enables log(n) complexity for insertions and lookups.
* Can be extended to an AVL Tree for balance and performance.

**3.2 HashMap**

Each seat number is mapped to a seat object using a HashMap. This provides constant-time access to seat details such as category, availability, and customer name.

**3.3 Stack**

Two separate stacks manage undo operations:

* bookingStack: Records each booking for undoing reservations.
* cancellationStack: Records each cancellation for undoing a cancellation.

**4. System Operations and Algorithms**

**4.1 Booking a Seat**

1. Input category and seat number.
2. Lookup category in BST.
3. Confirm seat availability via HashMap.
4. Reserve seat and push action to bookingStack.

**4.2 Cancelling a Seat**

1. Input seat number.
2. Check booking status via HashMap.
3. Cancel booking and push action to cancellationStack.

**4.3 Undo Operation**

1. If undoing booking: pop from bookingStack, mark seat available.
2. If undoing cancellation: pop from cancellationStack, re-book seat.

**4.4 Seat Lookup**

Use HashMap to retrieve and display details in O(1) time.

**5. Optimization Strategies**

* Use of AVL Tree ensures balanced BST for predictable performance.
* Limit stack size (e.g., last 10 operations) to control memory usage.
* Lazy deletion from tree or map for speed vs. memory trade-offs.

**6. Scalability Discussion**

* System performance tested with 10,000+ seat entries.
* HashMap and AVL Tree scale logarithmically or better.
* Stack operations are constant time and suitable for undo features.

**7. Sample Test Results**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Input** | **Output / Action** |
| Book Seat | Economy, A102 | Seat Booked Successfully |
| Cancel Seat | A102 | Seat Cancelled |
| Undo Booking | — | Seat A102 Released |
| Undo Cancellation | — | Seat A102 Re-booked |

**8. Potential Extensions**

* GUI integration for interactive seat maps.
* Time-based reservations and auto-expiry.
* Support for group booking.
* Seat priority allocation for VIP or loyalty customers.

**9. Conclusion**

This project effectively models a real-world ticket booking system using fundamental data structures. The use of BST, HashMap, and Stack provides a blend of speed, flexibility, and undo functionality. The system is modular and extensible for future enhancements.

**10. Deliverables**

**A. Written Report**

A 7–10 page document containing:

* Detailed problem description
* Justification for data structure choices
* Full explanation of algorithms
* Optimization and scalability strategies
* Test results and observations

**B. Program**

* Java files with complete implementation
* Sample dataset and test cases
* README with compile/run instructions and documentation

**C. Presentation**

* Slide deck covering design, data structures, flowcharts
* Demos (screenshots or live execution)
* Explanation of results and edge case handling

**Project Title 2: Job Portal with Candidate Shortlisting**

**1. Introduction**

Modern recruitment demands intelligent systems to efficiently manage job postings and candidate applications. This project models a Job Portal that handles employer job listings, candidate registrations, and shortlisting based on predefined filters such as skills, experience, and location. Implemented in Java, the system utilizes core data structures to ensure fast search, efficient matching, and real-time analytics for decision-making.

**2. Problem Definition**

The project aims to address the following challenges:

* Efficient management of job postings and candidate applications
* Intelligent shortlisting based on recruiter-defined filters
* Real-time ranking of candidates based on profile match
* Logging of application events for analytical insights

The system must be capable of handling thousands of users and simultaneous operations.

**3. Data Structures and Design**

**3.1 Circular Queue**

Used to manage interview scheduling slots in a First-In-First-Out (FIFO) manner.

* Ensures fair and efficient scheduling without duplication.

**3.2 HashMap**

Stores user profiles and job details for quick access and updates.

* **Key**: Unique ID (e.g., User ID, Job ID)
* **Value**: Profile or job object containing metadata

**3.3 Linked List**

Maintains an ordered log of all applications and interview events, supporting analytics such as most applied jobs, recruiter activity, and response time.

**3.4 Max-Heap (Priority Queue)**

Used to rank candidates based on profile relevance to job criteria.

* Ensures top-matching candidates are shortlisted first.

**4. System Operations and Algorithms**

**4.1 Job Application**

1. Candidate selects a job and applies.
2. Profile is evaluated and ranked using Max-Heap.
3. Application is stored in HashMap.
4. Log is appended to Linked List.

**4.2 Job Posting**

1. Recruiter posts a job with filters (skills, experience).
2. Job is added to HashMap.
3. Candidates are automatically evaluated for match.

**4.3 Shortlisting Engine**

1. Extract filters from job posting.
2. Use Max-Heap to rank candidates.
3. Notify top-ranked candidates.
4. Store actions in Linked List for analysis.

**5. Optimization Strategies**

* Use composite keys in HashMap for tracking multiple applications from the same user
* Leverage circular queues for efficient interview scheduling
* Use lazy updates in Max-Heap to optimize re-ranking during mass job postings
* Compress Linked List logs for long-term analytical storage

**6. Scalability Discussion**

* HashMap enables O(1) retrieval of job and user profiles
* Circular queues scale well for scheduling
* Max-Heap adapts efficiently to growing candidate pools
* Successfully tested with 10,000+ applications under concurrent usage

**7. Sample Test Results**

| **Operation** | **Input** | **Output / Action** |
| --- | --- | --- |
| Job Posting | Job: Backend Developer | Job ID: J101 added to system |
| Application | User: U256 applies to J101 | Ranked #2, Added to shortlist |
| Shortlisting | J101 | Top 5 candidates notified |
| Analytics | — | Most Applied Job: J101, Avg Rank: 3.2 |

**8. Potential Extensions**

* Resume parsing and scoring with AI
* Chatbot for candidate support
* Integration with LinkedIn for profile import
* Real-time recruiter dashboard with analytics
* Support for video interviews and scheduling

**9. Conclusion**

This project delivers a practical and scalable recruitment solution tailored for modern hiring workflows. By integrating data structures like circular queues, hashmaps, linked lists, and heaps, the system ensures optimized shortlisting and responsive operations. Its modular design enables easy integration of AI-powered analytics and future enhancements.

**10. Deliverables**

**A. Written Report**

* Detailed design and system architecture
* Explanation of algorithms with complexity analysis
* Justification for data structure selection
* Sample analytics and candidate match discussion

**B. Program**

* Modular Java classes implementing all functionalities
* README with deployment and usage instructions
* Test case files with execution results

**C. Presentation**

* Visual diagrams of system architecture and data flow
* Highlights of core features and outputs
* Future scope and enhancement roadmap

**Project Title 3: Smart Parking System with Analytics**

**1. Introduction**

Urban mobility and smart cities increasingly demand intelligent parking systems. This project models a Smart Parking System that tracks vehicle entry/exit, allocates slots dynamically, supports VIP prioritization, and logs activities for analytical insights. Implemented in Java, the system integrates core data structures to balance speed, accuracy, and real-time responsiveness.

**2. Problem Definition**

The project aims to address the following challenges:

* Real-time tracking of parking slot availability
* Efficient vehicle check-in/check-out
* Priority handling for VIP customers
* Entry/exit log generation for analytics

The system must be capable of supporting hundreds of vehicles and concurrent operations.

**3. Data Structures and Design**

**3.1 Circular Queue**

Used to manage available parking slots in a First-In-First-Out (FIFO) manner, especially for regular customers.

* Allows constant-time allocation and release of slots.

**3.2 HashMap**

Stores vehicle details (plate number, entry time, assigned slot, status) for quick lookup and updates.

* Key: Vehicle ID (e.g., plate number)
* Value: Vehicle object with metadata

**3.3 Linked List**

Maintains an ordered log of all entry and exit events, supporting analytics such as average stay duration and peak hours.

**3.4 Max-Heap (Priority Queue)**

Used to prioritize VIP vehicle entries.

* VIPs always get the best available slots

**4. System Operations and Algorithms**

**4.1 Vehicle Entry**

1. Check if vehicle is VIP.
2. Allocate slot from Max-Heap or Circular Queue.
3. Store entry in HashMap.
4. Append entry log to Linked List.

**4.2 Vehicle Exit**

1. Lookup vehicle in HashMap.
2. Remove from slot queue or heap.
3. Log exit time in Linked List.
4. Remove vehicle from map.

**4.3 Analytics Engine**

1. Process Linked List logs.
2. Generate reports on usage patterns, peak hours, and average stay durations.

**5. Optimization Strategies**

* Pre-allocate parking slots to reduce runtime checks
* Use doubly-linked list for faster reverse traversal
* Use composite keys in HashMap for managing duplicate entries (e.g., same plate over time)

**6. Scalability Discussion**

* Circular queue and hashmap scale well with O(1) operations.
* Heap and linked list offer efficient dynamic behavior.
* Successfully simulated for 1,000+ vehicles in stress testing.

**7. Sample Test Results**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Input** | **Output / Action** |
| Entry | VIP-9823 | Allocated VIP Slot V1 |
| Entry | REG-7842 | Allocated Slot R12 |
| Exit | REG-7842 | Freed Slot R12, Logged |
| Analytics | — | Peak Hour: 11 AM, Avg Stay: 2.3 hrs |

**8. Potential Extensions**

* Integration with payment gateway
* Mobile app for drivers
* Real-time map visualization of parking slots
* License plate recognition system (image processing)

**9. Conclusion**

The project demonstrates a practical and scalable solution for modern parking management. Leveraging circular queues, hashmaps, linked lists, and heaps, the system achieves real-time performance and insightful analytics. The design is modular and ready for real-world extensions.

**10. Deliverables**

**A. Written Report**

* Comprehensive design analysis
* Algorithm details with complexity assessment
* Data structure selection justification
* Sample analytics and discussion

**B. Program**

* Java classes with modular components
* README with setup instructions
* Test case files and documentation

**C. Presentation**

* Slides covering architecture, flow, and visuals
* Highlight features with sample outputs
* Discussion on results and future enhancements

**Project Title 4: Real-Time Leaderboard for Coding Contest**

**1. Introduction**

In competitive programming platforms and coding contests, leaderboards play a critical role in tracking participant performance in real-time. This project aims to build a dynamic leaderboard system using Java that efficiently processes participant submissions, updates scores, ranks contestants, and displays live rankings with optimized data structures.

**2. Problem Definition**

The objective is to maintain an up-to-date leaderboard that:

* Tracks individual user scores based on submissions
* Processes submissions in real-time order
* Dynamically ranks users based on score
* Resolves ties and allows for score updates

The system must handle high-frequency submission updates and scale with hundreds or thousands of contestants.

**3. Data Structures and Design**

**3.1 HashMap**

Stores user profile and their cumulative scores. Key: user ID, Value: score + metadata (e.g., time of last submission).

* Enables constant-time updates and lookups.

**3.2 Queue**

Models the sequence of user submissions as a FIFO structure to ensure real-time, chronological processing.

**3.3 Max-Heap (PriorityQueue)**

Maintains dynamic ranking of participants, with the highest score at the root.

* Allows real-time sorting and retrieval of top users.

**4. System Operations and Algorithms**

**4.1 Submission Processing**

1. Dequeue submission from the queue.
2. Retrieve and update user score in HashMap.
3. Update heap to maintain correct order.

**4.2 Leaderboard Display**

1. Copy and sort the heap using heap sort.
2. Display top N ranked users.

**4.3 Tie-Breaking**

1. In case of equal scores, rank by earliest submission.
2. Maintain timestamps within user metadata.

**4.4 Score Update Algorithm**

Use an update-in-place mechanism with heap adjustment logic to maintain priority order.

**5. Optimization Strategies**

* Use Indexed Priority Queue for more efficient score updates.
* Batched re-heapification for bulk updates.
* Thread-safe queues for concurrent processing.

**6. Scalability Discussion**

* Designed to support 10,000+ users with minimal latency.
* Heaps and maps allow logarithmic or constant-time operations.
* Performance tested using synthetic load generators.

**7. Sample Test Results**

| **User ID** | **Submissions** | **Final Score** | **Rank** |
| --- | --- | --- | --- |
| u101 | 5 | 480 | 1 |
| u235 | 6 | 450 | 2 |
| u012 | 4 | 450 | 3\* |

(\*) Tie resolved by earliest submission time.

**8. Potential Extensions**

* Multi-problem contest support
* Penalty for wrong submissions
* Rating history tracking per user
* Web-based UI for real-time leaderboard

**9. Conclusion**

The project successfully demonstrates an efficient, scalable solution for maintaining a real-time leaderboard in coding competitions. By combining HashMaps, Queues, and Heaps, the system ensures responsiveness and fairness, even under heavy loads. With optional extensions, it can evolve into a complete contest management engine.

**10. Deliverables**

**A. Written Report**

* Full explanation of system goals, design choices, and data structures
* Algorithm walkthroughs with complexity analysis
* Scalability tests and results

**B. Program**

* Java source files with structured modules
* Test cases and README for compilation and usage
* Inline documentation for maintainability

**C. Presentation**

* Conceptual diagrams
* Code architecture overview
* Result analysis and visualizations
* Live demo walkthrough or screenshots

**Project Title 5: Smart Inventory Management System**

**1. Project Overview**

The Smart Inventory Management System is a Java-based application designed to efficiently manage a large dataset of inventory items using tree data structures. It simulates a real-world inventory system, supporting key operations such as adding, deleting, updating, and querying items.

This project emphasizes the use of **data structures and algorithms**, especially:

* **Tree data structures** for organizing items
* **File I/O** for persistent backup and recovery
* **Undo/Redo mechanisms** for operation traceability and reliability

The system is scalable and optimized to handle thousands of inventory records and supports basic and advanced search features.

**2. System Components**

**Inventory Items**

Each item in the inventory includes:

* **Item ID** (unique integer key)
* **Name** (string)
* **Category** (e.g., electronics, stationery)
* **Price** (floating point)
* **Quantity** (integer)

**3. Data Structure Design**

**Tree Structure**

A **tree-based data structure** (e.g. Balanced Tree) is used to:

* Store and organize items based on Item ID
* Enable efficient in-order traversal for sorted views
* Support fast insertion, deletion, and search operations

Tree nodes will contain Item objects, and tree traversal will support operations like:

* Search by ID
* Sorted display by ID or Name
* Low-stock item filtering

**4. File I/O Functionality (Mandatory)**

**Loading Dataset**

At program start, the system will:

* Automatically **read from a text file** (inventory\_backup.txt)
* Parse each line to reconstruct the inventory into the tree

**Saving Backup**

At program exit or on user request, the system will:

* **Serialize and write the tree** to a file
* Store the inventory in a readable format:

yaml

CopyEdit

1001,Notebook,Stationery,120.50,30

1002,Pen,Stationery,25.00,200

This ensures data persistence and easy recovery.

**5. Undo/Redo Functionality (Mandatory)**

The system must support full **Undo and Redo** capabilities for the following operations:

* Insert
* Delete
* Update

**Design Requirements**

* Maintain operation history using **Stacks** or other efficient structures (e.g., Deques, Command pattern).
* Undo will reverse the most recent change.
* Redo will reapply a recently undone change.
* Each action should be recorded with enough metadata to be reversed.

**6. Large Dataset Requirement**

The system must demonstrate performance on a **large dataset of at least 1000 inventory records**. This will help evaluate:

* Time complexity of operations
* Tree depth and balance
* Backup file load/save time
* Undo/redo performance over long sessions

**7. Search & Display Features**

The system must allow:

* Search by **ID**
* Search by **Name or Category**
* Search by **Price Range**
* Display sorted inventory (by ID or Name)
* Display **low-stock items** under a given quantity threshold

**8. Evaluation & Report Guidelines**

Your final submission must include:

* Java code files (
* Sample backup file (≥1000 items)
* Project report (2–4 pages) covering:
  + Tree structure design
  + Undo/Redo mechanism and data structure justification
  + File I/O design and format
  + Time/space complexity of key operations
  + Sample results with large data
  + Scalability and optimization ideas

**9. Optional Features (Bonus Points)**

* Auto-save every X operations
* Analytics: total inventory value, average item price