VIDEO RENTING SYSTEM

Name: Hudhayfa Jeilani
Role: Project manager
Student Id: M00847160
Github user: hjhudz

Name: Adan Alaso
Role: Developer
Student Id: M00871832
Github user: aa5405

Name: Hussein Al Wazny
Role: Tester
Student Id: M00913787
Github user: Hussein129-1

Name: Abdulkadir Hussein
Role: Secretary
Student Id: M00873978
Github user: ab-hussein

1. Introduction

The Video Renting System is a console application that effectively deals with video rentals using optimized custom data structures and an SQL database. The system consists of admin and customer features (handling customers, videos, rentals, and users) and (video renting and returning).

Key Features

a. User Authentication
   - Role-based access (Admin/Customer).
   - Secure login using UserList (linked list).

b. Video Management
   - Binary Search Tree (BST) for storing videos (VideoBST), enabling efficient $O(\log n)$ search, insertion, and deletion.
   - Admins can add, remove, update, and display videos.

c. Customer Management
   - Linked List (CustomerList) for storing customer data, supporting $O(1)$ insertion but $O(n)$ search (acceptable for small datasets).

d. Rental Management
   - Hash Table (RentalHashTable) for tracking rentals, providing $O(1)$ average-time complexity for adding and returning rentals.
   - Customers can rent/return videos, while admins monitor pending returns.

e. Database Integration
   - SQL database stores persistent data (videos, customers, rentals, users).
   - Data is loaded at startup (LoadData()) and updated after modifications.

Technical Implementation

The system uses:

- Hash Table (RentalHashTable) for fast rental operations.
- BST (VideoBST) for efficient video searches.
- Linked Lists (CustomerList, UserList) for simpler user/customer management.

2. Design

Justification of Data Structures

The Video Renting System employs custom data structures to optimize performance for key operations. Below is the rationale behind each choice:

a. Binary Search Tree (BST) for Video Management (VideoBST)

- Why BST?
  - **Fast Search:** $O(\log n)$ average time complexity for searching videos by ID.
  - **Ordered Storage:** Maintains videos in sorted order (by ID), enabling efficient range queries.
- Use Case:
  - Frequent video searches (e.g., checking availability before rental).
  - Admin operations (add/remove/update videos).

- Alternative Considered:
  - Hash Table: Rejected because it does not maintain order, making sorted displays (e.g., "Display All Videos") inefficient.

b. Hash Table for Rental Management (RentalHashTable)

- Why Hash Table?
  - $O(1)$ Average-Time Operations: Inserting, searching, and updating rentals are optimized.
  - Collision Handling: Uses chaining (linked lists) to manage hash collisions.
- Use Case:
  - Tracking rentals (new rentals, returns, pending returns).
  - Quickly accessing rental records by ID.
- Alternative Considered:
  - Linked List: Rejected due to $O(n)$ search time, which is inefficient for large rental datasets.

c. Linked List for Customer & User Management (CustomerList, UserList)

- Why Linked List?
  - Simple Implementation: Easy to maintain for small datasets.
  - $O(1)$ Insertions: New customers/users are added at the head.
- Use Case:
  - Less frequent operations (e.g., customer updates, user authentication).
- Alternative Considered:
  - BST: Rejected because the added complexity was unnecessary for small user/customer lists.

Analysis of Key Algorithms

a. Renting a Video

Pseudo Code:

Algorithm RentVideo(customerID, videoID):
1. Search customer in CustomerList → $O(n)$
2. Search video in VideoBST → $O\ (\log n)$
3. If both found and video available:
   3.1 Add rental record in RentalHashTable → $O(1)$
   3.2 Update video status in VideoBST → $O\ (\log n)$
4. Return success/failure

Overall Time Complexity: $O(n)$ (CustomerList dominates)

Justification:

- The $O(n)$ customer search is the bottleneck but acceptable since customer operations are less frequent.
- Video search and rental insertion are optimized ($O\ (\log n)$) and $O(1)$)

b. Returning a Video

Pseudo Code:

```
Algorithm ReturnVideo(rentalID):
  1. Search rental in RentalHashTable → O(1)
  2. Update rental status → O(1)
  3. Update video availability in VideoBST → O (log n)
Overall Time Complexity: O (log n)
```

Justification:

- Hash table lookup ensures fast rental retrieval.
- Video status update is efficient due to BST.

c. Searching for a Video

Pseudo Code:

```
Algorithm SearchVideo(videoID):
  1. Search VideoBST → O (log n)
  2. Return video details if found
Overall Time Complexity: O (log n)
```

Justification:

- BST enables efficient searches, critical for frequent video queries.

d. Adding a New Customer

Pseudo Code:

```
Algorithm AddCustomer(customer):
  1. Insert at head of CustomerList → O(1)
Overall Time Complexity: O(1)
```

Justification:

- Linked list allows O(1) insertions, making admin operations fast.

3. Testing

Testing Approach

The Video Rented System was exhaustively tested via **Microsoft.VisualStudio.TestTools.UnitTesting** for accuracy and dependability. The testing strategy included:

- **Unit Testing:** All the data structures (CustomerList, UserList, RentalHashTable, VideoBST) were unit tested individually.

- **Integration Testing:** Verified interactions between components (e.g., video rental updates both RentalHashTable and VideoBST).
- **Edge Cases:** Tested boundary conditions (e.g., empty lists, invalid IDs).

Test Cases

CustomerList Test Cases

Test Case	Description	Input	Expected Result	Actual Result	Pass/Fail
Instance_ShouldReturnSingletonInstance	Verify singleton pattern works correctly	Access Instance twice	Both instances are the same	Same reference returned	Pass
AddCustomer_ShouldAddCustomerToTheList	Add single customer to empty list	testCustomer1	Count = 1	Count = 1	Pass
AddCustomer_ShouldAddMultipleCustomers	Add 3 customers sequentially	testCustomer1, testCustomer2, testCustomer3	Count = 3	Count = 3	Pass
RemoveCustomer_ShouldRemoveExistingCustomer	Remove existing customer (ID=1)	List with 2 customers → Remove ID=1	true, Count = 1	true, Count = 1	Pass
RemoveCustomer_ShouldReturnFalseForNonExistingCustomer	Attempt to remove invalid ID (999)	List with 1 customer → Remove ID=999	false, Count unchanged	false, Count=1	Pass

Key Observations

a. Functionality Verified:
   - Singleton pattern enforcement.
   - CRUD operations (Add/Remove/Update/Clear) work as expected.
   - Edge cases (empty list, invalid IDs) handled correctly.

b. Performance Notes:
   - AddCustomer: Confirmed $O(1)$time complexity (prepend operation).
   - Remove/Update: O(n) time complexity due to sequential search (expected for linked list).

c. Limitations:
   - No stress testing for large datasets (>10,000 customers).
   - No thread-safety tests for concurrent access.

UserList Test Cases

Test Case	Description	Input	Expected Result	Actual Result	Pass/Fail
Instance_ShouldReturnSingletonInstance	Verify singleton pattern	Access Instance twice	Both instances same	Same reference returned	Pass
AddUser_ShouldAddUserToTheList	Add admin user to empty list	adminUser	User retrievable via GetUser	User found	Pass
GetUser_ShouldReturnCorrectUser	Retrieve existing user	Username="customer1"	Returns correct user object	Correct user returned	Pass
GetUser_ShouldReturnNullForNonExistingUser	Attempt to retrieve invalid user	Username="nonexistent"	Returns null	null returned	Pass
ValidateUser_ShouldReturnUserForValidCredentials	Authenticate with correct credentials	Username="admin", Password="admin123"	Returns user object	User returned	Pass

Key Observations

a. Core Functionality Verified:
   - User authentication (ValidateUser) works correctly with valid/invalid credentials
   - Role verification (IsAdmin, IsCustomer) functions as expected
   - CRUD operations (Add/Remove/Update/Clear) perform correctly
   - Singleton pattern enforced for UserList

b. Edge Cases Handled:
   - Invalid usernames/passwords
   - Non-existent user IDs
   - Empty list scenarios

c. Performance Notes:
   - AddUser: $O(1)$time complexity (prepend operation)
   - GetUser/RemoveUser/UpdateUser: $O(n)$time complexity (sequential search)
   - Authentication checks execute in $O(n)$ time

VideoBST Test Cases

Test Case	Description	Input	Expected Result	Actual Result	Pass/Fail
Instance_ShouldReturnSingletonInstance	Verify singleton pattern	Access Instance twice	Both instances same	Same reference returned	Pass
Insert_ShouldAddVideoToBST	Insert single video	video1 (ID=10)	Video retrievable via Search	Video found	Pass

Insert_ShouldMaintainBSTProperty	Insert 4 videos in mixed order	IDs: 10, 5, 15, 7	In-order traversal returns sorted IDs: 5,7,10,15	Correct order returned	Pass
Search_ShouldFindExistingVideo	Search for inserted video	ID=5 (video2)	Returns correct video	Correct video returned	Pass
Search_ShouldReturnNullForNonExistingVideo	Search for invalid ID	ID=999	Returns null	null returned	Pass

Key Observations

a. BST Properties Verified:
   - Insertion maintains proper ordering (left < parent < right)
   - In-order traversal returns sorted results
   - Deletion handles all cases (0, 1, and 2 children) correctly

b. Performance Confirmed:
   - Insert/Search/Update/Delete: $O\ (\log n)$ average case performance
   - Worst Case: $O(n)$ when tree becomes unbalanced (not tested)

c. Edge Cases Handled:
   - Empty tree operations
   - Invalid video IDs
   - Boundary conditions (min/max IDs)

RentalHashTable Test Cases

Test Case	**Description**	**Input**	**Expected Result**	**Pass/Fail**
Instance_ShouldReturnSingletonInstance	Verify singleton pattern	Access Instance twice	Both instances same	Pass
AddRental_ShouldAddNewRental	Add single rental	rental1	Count increases to 1	Pass
AddRental_ShouldSetDefaultStatus	Add rental without status	Status=null	Defaults to "Rented"	Pass
AddRental_ShouldAutoIncrementRentalID	Add 3 rentals	Sequential adds	IDs: 1, 2, 3	Pass

GetRental_ShouldReturnCorrectRental	Retrieve existing rental	Valid RentalID	Returns correct rental	Pass

Key Observations

a. Core Operations:
   - Add/retrieve/return rentals with $O(1)$ average time complexity
   - Automatic ID generation
   - Default status assignment
b. Counting Features:
   - Total rentals count
   - Pending returns (all/customer-specific)
c. Edge Cases:
   - Non-existent rental IDs
   - Empty table handling
   - Null/default values

4. Conclusion

Video Renting System successfully employs key features with fine-tuned data structures, making it efficient for managing video rentals, customer data, and user authentication. The key achievements and limitations are:

key achievements

a. Data Structures Implemented:
   - Binary Search Tree (BST) for video management : O $(\log n)$ search/insert/delete).
   - Hash Table for rental transactions ($O(1)$ average-time operations).
   - Linked Lists for customers and users (simple CRUD operations).

b. Key Features Achieved:
   - Admin Functions: Add/remove videos, manage rentals, and view pending returns.
   - Customer Functions: Rent/return videos with real-time availability updates.
   - User Authentication: Role-based access control (Admin/Customer).

Limitations

a. Scalability Issues:
   - CustomerList uses O(n) search time (linked list) – inefficient for large datasets.
   - BST could degrade to O(n) if unbalanced (no auto-balancing implemented).
b. Missing Features:
   - No password encryption (stores plaintext hashes).
   - No bulk import/export for rentals or videos.
c. Testing Gaps:
   - No stress tests for >10,000 entries.
   - No concurrency tests for multi-user environments.