Lecture 7 -Data Structures and Database Design

IST 3108 Application Development

Dr. Peter Khisa Wakholi

Dept of Information Systems, Makerere University

Online Resources

- Online tutorials and resources for data structures and database design:
  - W3Schools Database Design
  - GeeksforGeeks Data Structures

Agenda

- Overview of data structures.
- Data structures in JavaScript.
- Principles of good database design
- Task 7 in Detail

Data Structures

Part 1

Overview of Data Structures

Data structures are fundamental components for organizing and storing data efficiently.

They provide a way to manage, access, and manipulate data.

Essential for improving algorithm efficiency and solving real-world problems.

Importance of Data Structures

Data structures affect how data is stored and accessed, which can significantly impact the performance of an application.

Choose the right data structure for specific use cases to optimize performance and maintainability.

Data Structures in JavaScript

- JavaScript offers several built-in data structures:
    - Arrays
    - Objects
    - Sets
    - Maps
    - Stacks and Queues
    - Linked Lists
    - Trees and Graphs

Arrays in JavaScript

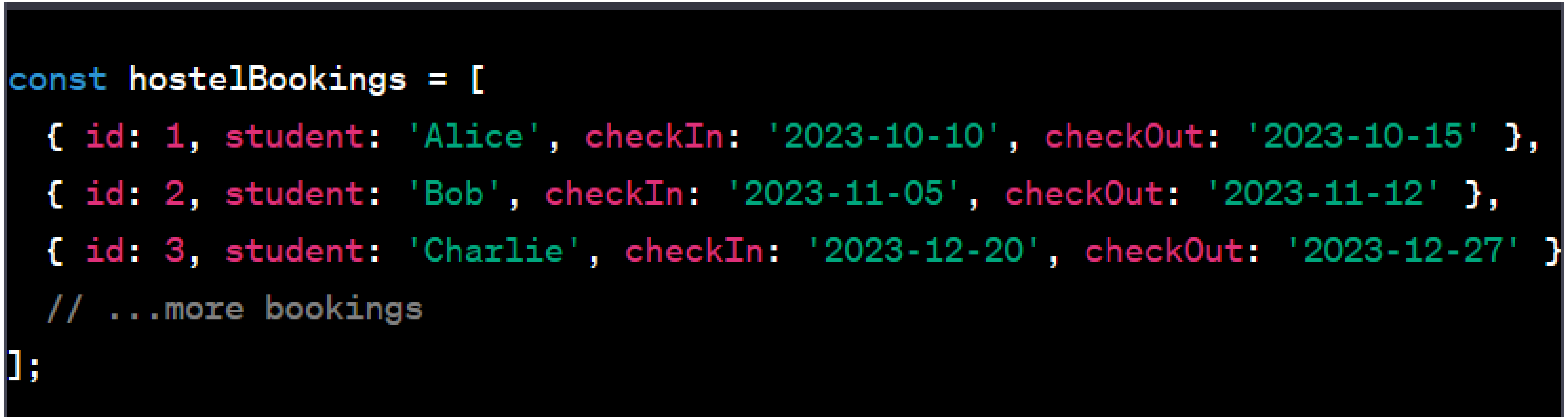
Arrays are ordered lists used to store multiple values.

Access elements using indices (0-based).

Common operations: adding, removing, and searching for elements.

Array Use Case: Managing Hostel Bookings

- Suppose we have an array of hostel bookings, where each booking is represented as an object with details like the booking ID, student name, check-in date, and check-out date.



```
const hostelBookings = [
  { id: 1, student: 'Alice', checkIn: '2023-10-10', checkOut: '2023-10-15' },
  { id: 2, student: 'Bob', checkIn: '2023-11-05', checkOut: '2023-11-12' },
  { id: 3, student: 'Charlie', checkIn: '2023-12-20', checkOut: '2023-12-27' }
  // ...more bookings
];
```

Objects in JavaScript

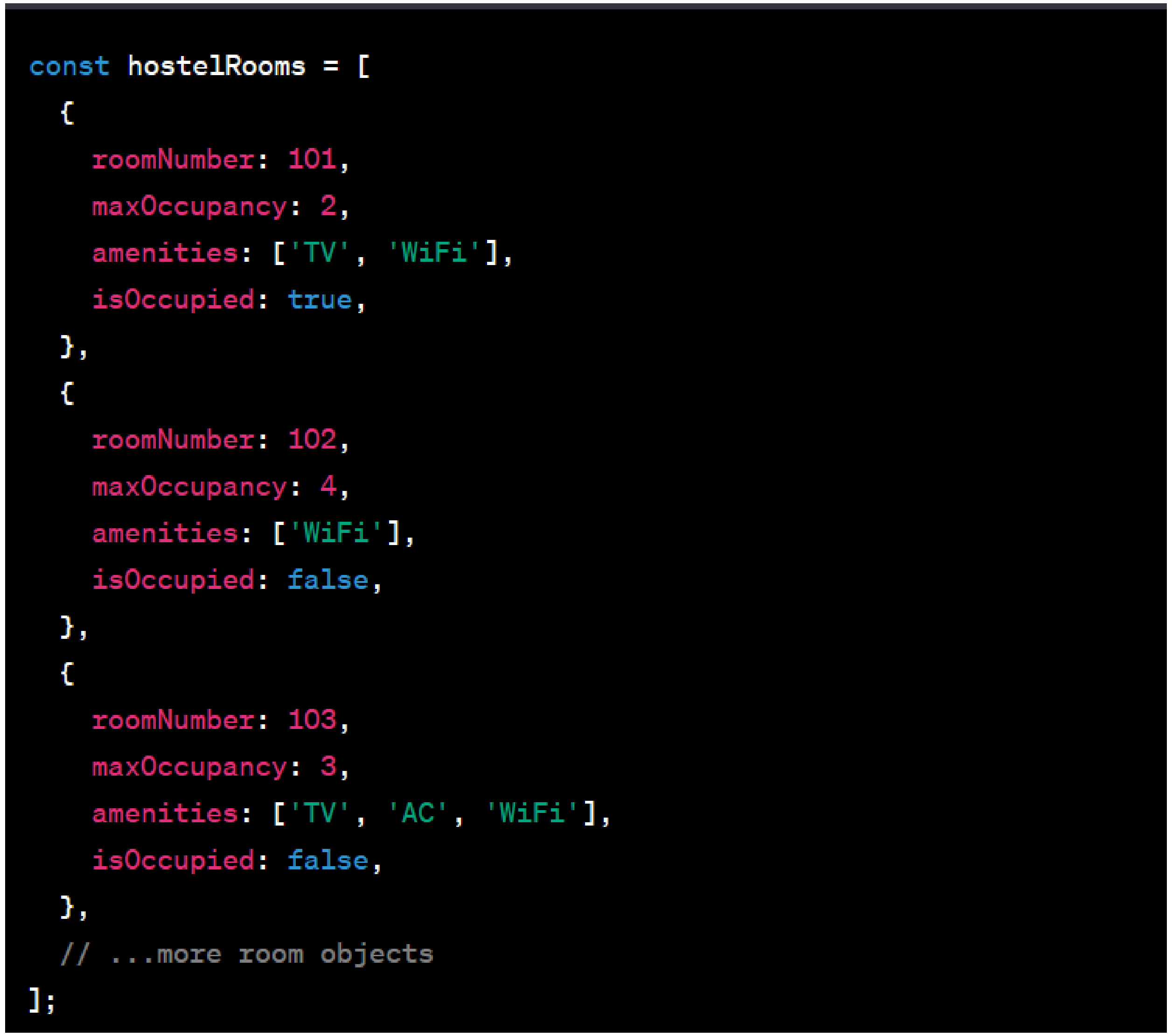
Objects are key-value pairs.

Versatile data structure for storing and organizing data.

Access properties using keys.

Objects Use Case: Managing Hostel Room Data

- Suppose we want to keep track of information about hostel rooms, such as room numbers, the maximum occupancy, and available amenities. We'll use JavaScript objects to represent this data.



```
const hostelRooms = [
  {
    roomNumber: 101,
    maxOccupancy: 2,
    amenities: ['TV', 'WiFi'],
    isOccupied: true,
  },
  {
    roomNumber: 102,
    maxOccupancy: 4,
    amenities: ['WiFi'],
    isOccupied: false,
  },
  {
    roomNumber: 103,
    maxOccupancy: 3,
    amenities: ['TV', 'AC', 'WiFi'],
    isOccupied: false,
  },
  // ...more room objects
];
```

Sets and Maps in JavaScript

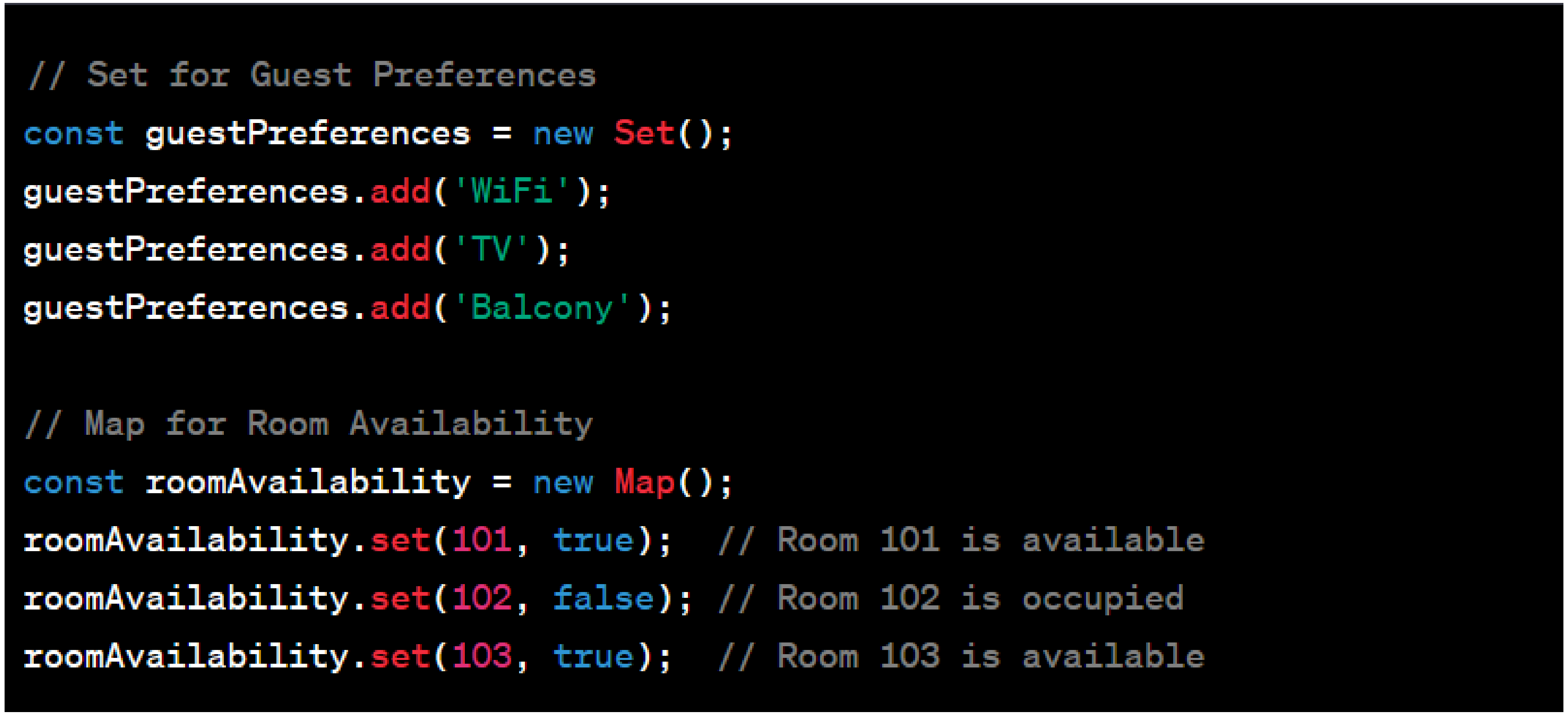
Sets store unique values (no duplicates).

Maps store key-value pairs.

Efficient for searching and retrieval.

Use Case: Managing Guest Preferences and Room Availability

- Suppose you want to keep track of guest preferences for room amenities and also track the availability of rooms. You can use Sets for guest preferences and Maps to manage room availability.



```
// Set for Guest Preferences
const guestPreferences = new Set();
guestPreferences.add('WiFi');
guestPreferences.add('TV');
guestPreferences.add('Balcony');

// Map for Room Availability
const roomAvailability = new Map();
roomAvailability.set(101, true);  // Room 101 is available
roomAvailability.set(102, false); // Room 102 is occupied
roomAvailability.set(103, true);  // Room 103 is available
```

Stacks and Queues in JavaScript

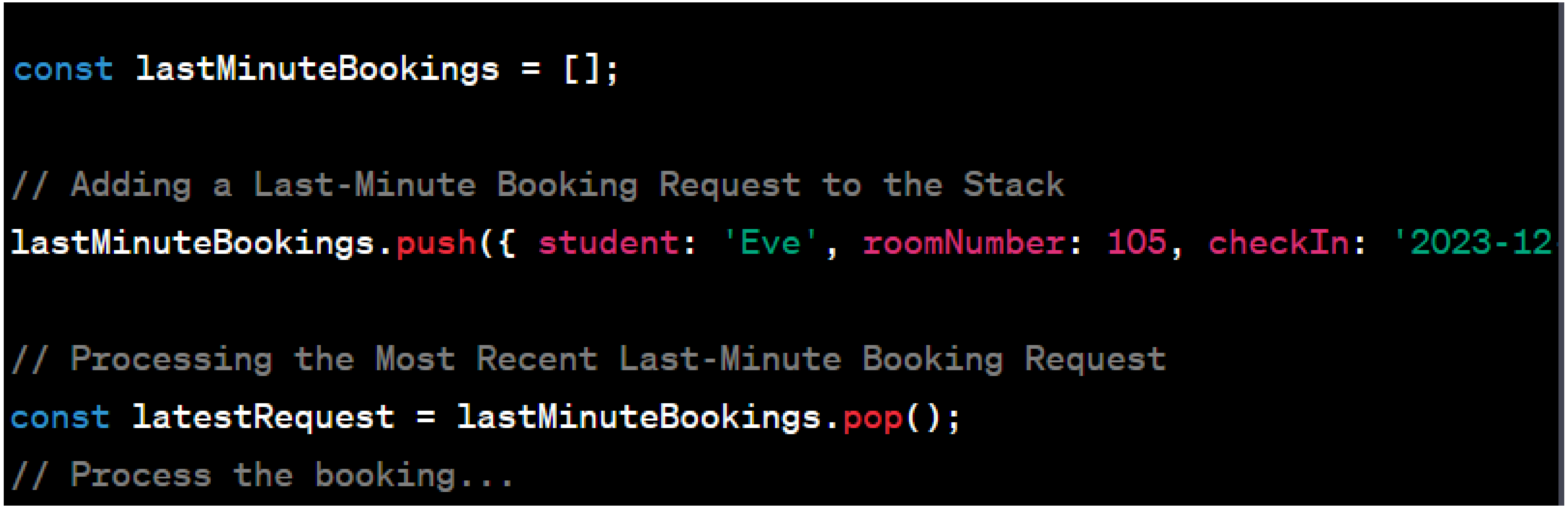
Stacks follow the Last-In-First-Out (LIFO) principle.

Queues follow the First-In-First-Out (FIFO) principle.

Commonly used for managing tasks and asynchronous operations.

Use Case: Managing Last-Minute Bookings with a Stack and Booking Requests with a Queue

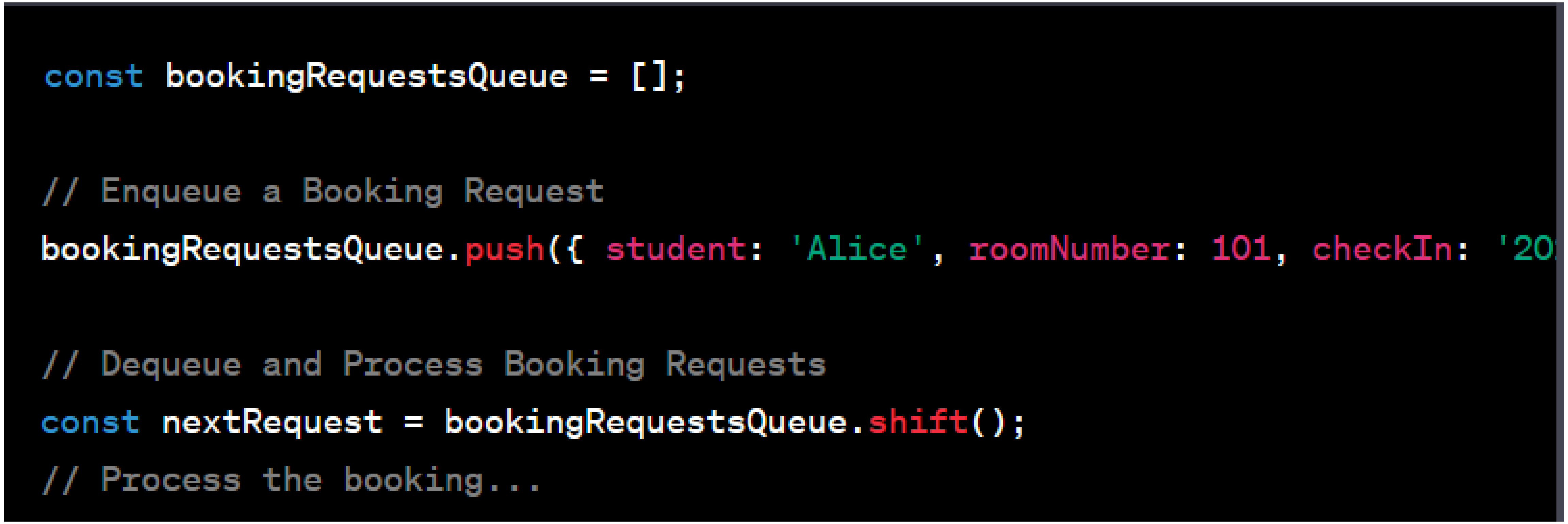
- Stack for Last-Minute Bookings: Suppose you want to manage last-minute bookings separately using a stack data structure. The stack will store booking requests for rooms, and the most recent requests will be processed first.



```
const lastMinuteBookings = [];

// Adding a Last-Minute Booking Request to the Stack
lastMinuteBookings.push({ student: 'Eve', roomNumber: 105, checkIn: '2023-12

// Processing the Most Recent Last-Minute Booking Request
const latestRequest = lastMinuteBookings.pop();
// Process the booking...
```

Queue for Booking Requests:

- You can manage regular booking requests using a queue data structure. The queue will ensure that booking requests are processed in a first-come, first-served manner.



```
const bookingRequestsQueue = [];

// Enqueue a Booking Request
bookingRequestsQueue.push({ student: 'Alice', roomNumber: 101, checkIn: '20

// Dequeue and Process Booking Requests
const nextRequest = bookingRequestsQueue.shift();
// Process the booking...
```

Linked Lists in JavaScript

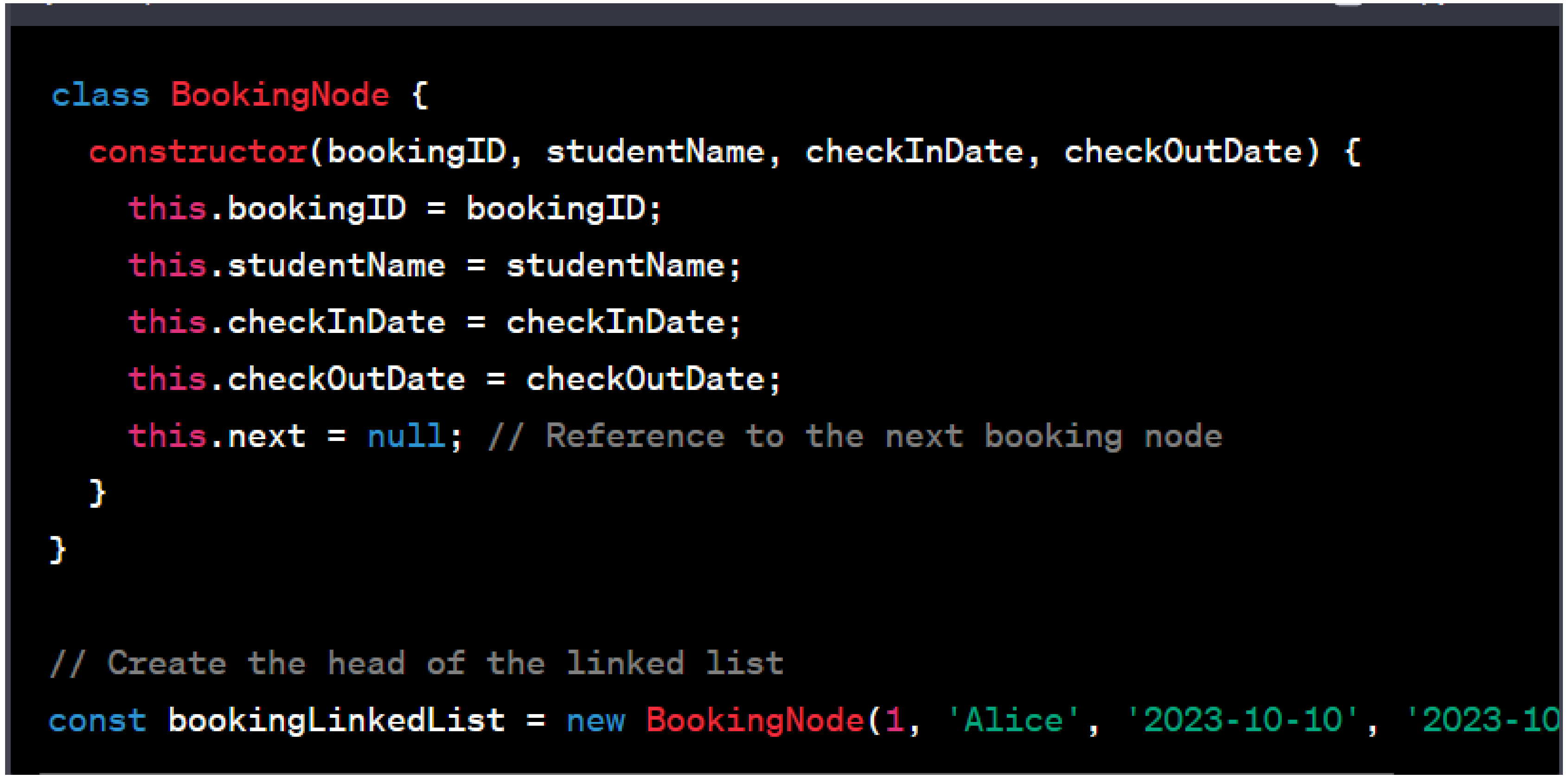
Linked lists are data structures with nodes connected by references.

Useful for dynamic data structures.

Types: singly linked list, doubly linked list.

Use Case: Managing Hostel Booking Linked List

- Suppose we want to manage a linked list of hostel bookings, where each booking node contains details about the booking, such as the booking ID, student name, check-in date, and check-out date.



```
class BookingNode {
  constructor(bookingID, studentName, checkInDate, checkOutDate) {
    this.bookingID = bookingID;
    this.studentName = studentName;
    this.checkInDate = checkInDate;
    this.checkOutDate = checkOutDate;
    this.next = null; // Reference to the next booking node
  }
}

// Create the head of the linked list
const bookingLinkedList = new BookingNode(1, 'Alice', '2023-10-10', '2023-10
```

Trees and Graphs in JavaScript

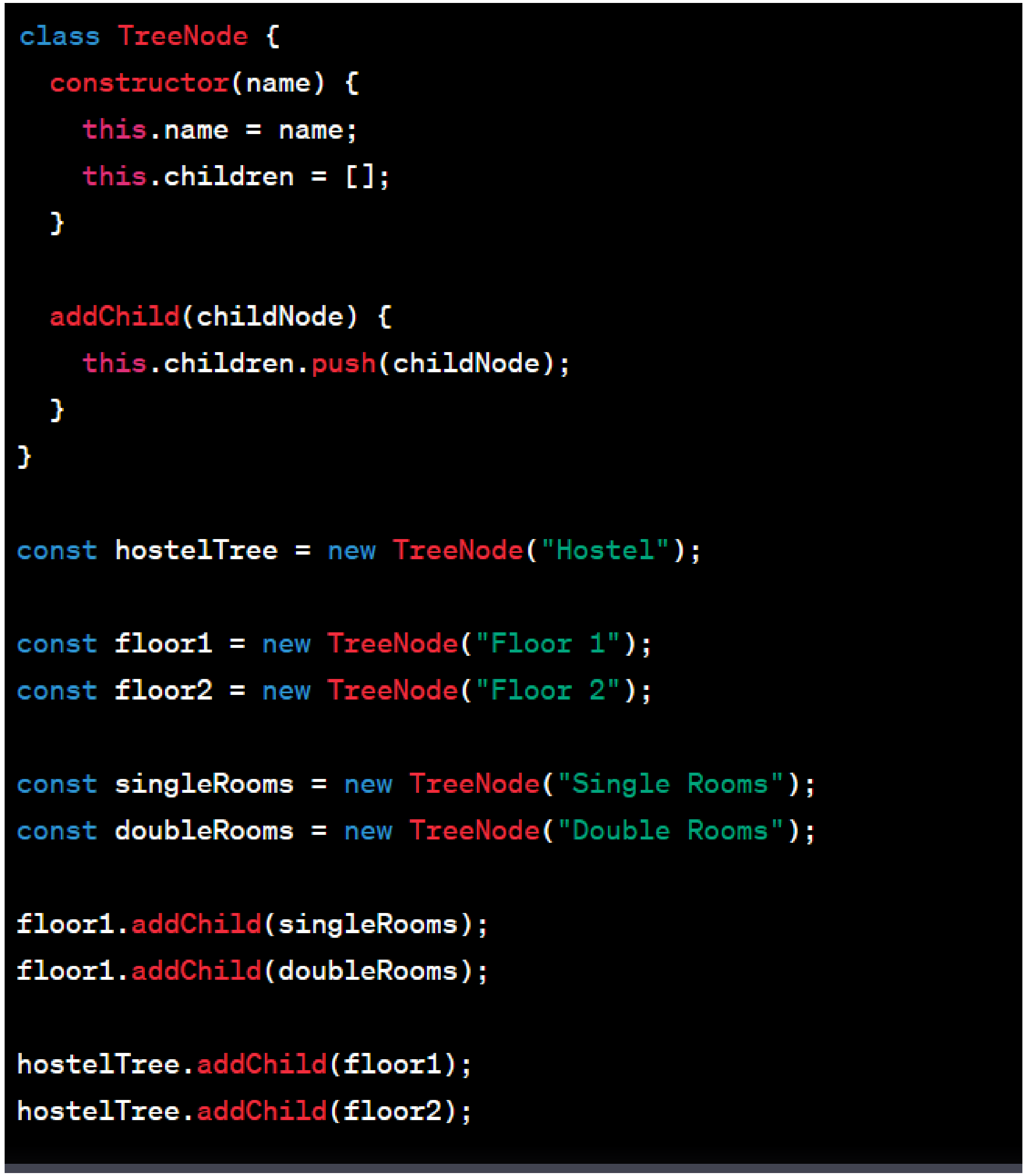
Trees and graphs are hierarchical data structures.

Trees have a single root node and child nodes.

Graphs have nodes connected by edges.

Tree Use Case: Representing Room Hierarchies with Trees

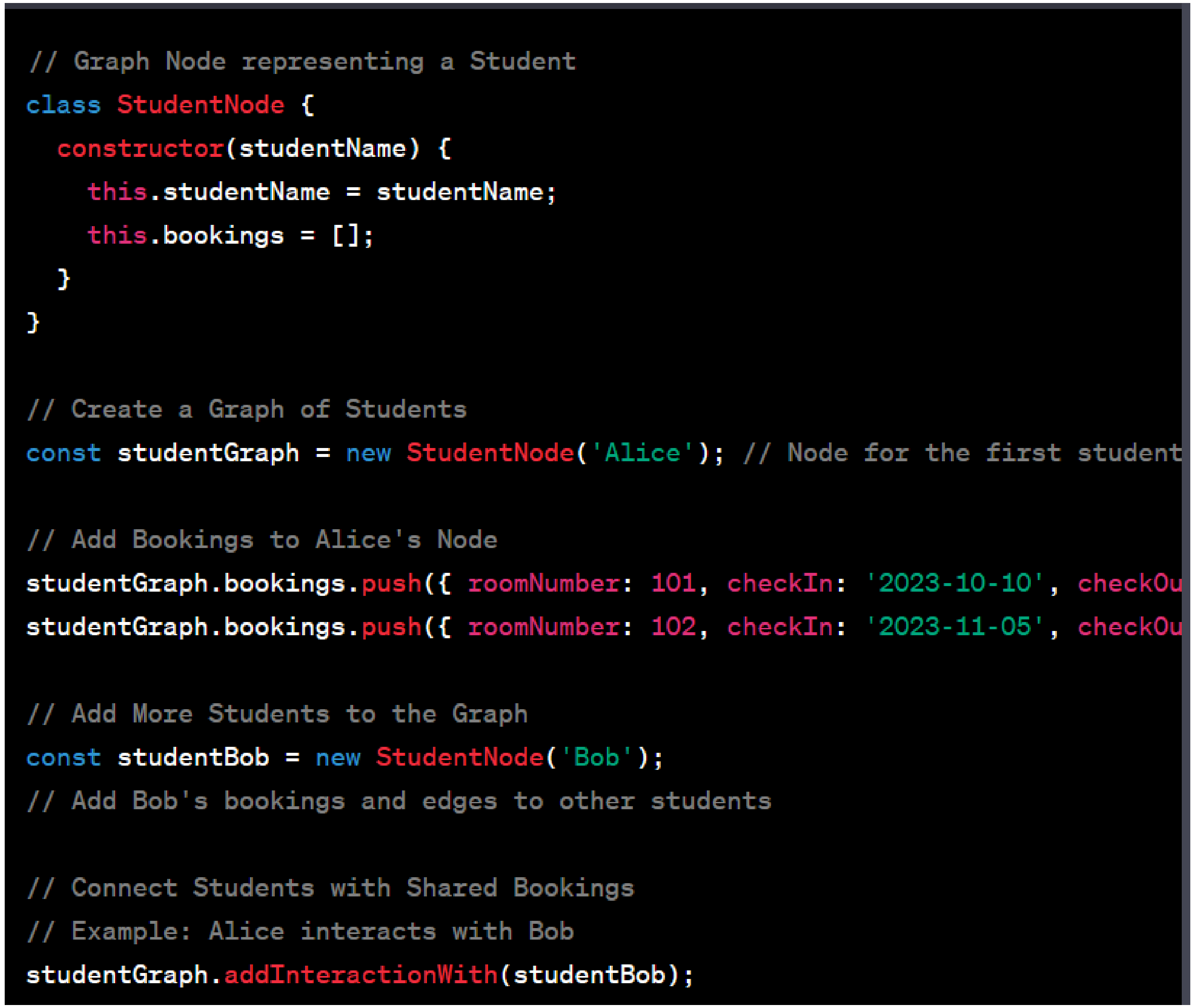
- Suppose you want to represent the hierarchical structure of rooms in a hostel. You can use a tree data structure to organize rooms into categories, such as floors or room types.



```
class TreeNode {
  constructor(name) {
    this.name = name;
    this.children = [];
  }

  addChild(childNode) {
    this.children.push(childNode);
  }
}

const hostelTree = new TreeNode("Hostel");

const floor1 = new TreeNode("Floor 1");
const floor2 = new TreeNode("Floor 2");

const singleRooms = new TreeNode("Single Rooms");
const doubleRooms = new TreeNode("Double Rooms");

floor1.addChild(singleRooms);
floor1.addChild(doubleRooms);

hostelTree.addChild(floor1);
hostelTree.addChild(floor2);
```

Graph Use Case: Representing Booking Interactions with a Graph

- You can represent interactions between students who make bookings using a graph. Each node represents a student, and edges between nodes indicate interactions or shared bookings.



```
// Graph Node representing a Student
class StudentNode {
  constructor(studentName) {
    this.studentName = studentName;
    this.bookings = [];
  }
}

// Create a Graph of Students
const studentGraph = new StudentNode('Alice'); // Node for the first student

// Add Bookings to Alice's Node
studentGraph.bookings.push({ roomNumber: 101, checkIn: '2023-10-10', checkOu
studentGraph.bookings.push({ roomNumber: 102, checkIn: '2023-11-05', checkOu

// Add More Students to the Graph
const studentBob = new StudentNode('Bob');
// Add Bob's bookings and edges to other students

// Connect Students with Shared Bookings
// Example: Alice interacts with Bob
studentGraph.addInteractionWith(studentBob);
```

Database Design Principles

Part 2

Database Design Principles

- Database design is a critical aspect of software development.
- Well-designed databases ensure data integrity and performance.
- Database design principles
  - normalization,
  - relationships,
  - entity-attribute-value,
  - data integrity,
  - referential integrity,
  - constraints,
  - auditing,
  - role-based access control

Use-Case 1: User Registration

Use Case 1: User Registration

Data Models:

- User table with fields like **user_id, username, email, password, first_name, last_name**.
- Booking table with fields like **booking_id, user_id, hostel_id, check_in_date, check_out_date.**

Principles Illustrated:

- **Normalization:** Storing user information in a separate table to avoid data redundancy.
- **Relationships:** Creating a relationship between users and their bookings using the **user_id** field in the Booking table.

Usecase 2: Hostel Listing

Data Models:

- Hostel table with fields like **hostel_id, hostel_name, location, price_per_night, availability**.

Principles Illustrated:

- **Entity-Attribute-Value (EAV):** Each hostel listing has various attributes (name, location, price, availability) stored in a structured way.
- **Data Integrity:** Ensuring data integrity by having a price_per_night field for accurate pricing.

Use Case 3: Booking Confirmation

Data Models:

Booking table with fields like **booking_id, user_id, hostel_id, check_in_date, check_out_date**.

Principles Illustrated:

Referential Integrity: Ensuring that a booking is linked to an existing user and hostel through the **user_id and hostel_id foreign keys.**

Constraints: Applying constraints to check that check-in and check-out dates are valid and do not overlap with other bookings.

Use Case 4: Payment Processing

Data Models:

- Payment table with fields like **payment_id, user_id, booking_id, amount, payment_date**.

Principles Illustrated:

- **Auditing:** Keeping a record of all payments made for transparency and auditing purposes.
- **Relational Design:** Linking payments to bookings through **booking_id** and to users through **user_id**.

Use Case 5: Admin Management

Data Models:

- Admin table for hostel owners and university staff with fields like **admin_id, username, password**.

Principles Illustrated:

- **Role-Based Access Control:** Having a separate Admin table for managing admin users with specific roles.
- **Security:** Safeguarding admin credentials through encryption and secure storage.

Task 7 - Overview

Guidelines

Task 7 Objective:

Design the database schema for the Hostel Booking App.

Develop data storage and retrieval functions for hostels and bookings.

Data base Schema

The database schema is the foundation of the Hostel Booking App's data management system.

Apply the principles of Good Database Design:

Normalization: Organize data to avoid redundancy and inconsistency.

Entity-Relationship Diagrams (ERD): Visualize the structure and relationships of data.

Data Integrity: Ensure data accuracy, consistency, and reliability.

Considerations:

Create tables for Hostels, Users, and Bookings.

Define relationships between tables using foreign keys (user_id, hostel_id, etc.).

Normalize data to minimize data duplication.

Data Storage Functions:

1

Develop functions for storing and retrieving data in the database:

- Insert new hostels and user information.
- Create new bookings and confirmations.
- Retrieve hostel listings and user bookings.

2

Utilize structured APIs and SQL queries.

Database Schema Diagram:

1

Display an ERD illustrating the relationships between the User, Hostel, and Booking tables, etc.

2

Highlight the use of foreign keys to connect these tables.

3

Emphasize how the schema reflects data integrity and the flow of information.