SMART PARKING LOT SYSTEM

23IT452–Operating Systems Laboratory

MINI PROJECT REPORT

SUBMITTED BY,

SHREE HARINI.K	(9517202306092)
SOWMIYA SHIVANI.S	(9517202306096)
VARSHA.G.A	(9517202306111)

DEPARTMENT OF INFORMATION TECHNOLOGY

MEPCO SCHLENK ENGINEERING COLLEGE, SIVAKASI

(An Autonomous Institution affiliated to Anna University Chennai)

2024- 2025

CERTIFICATE

This is to Certify that it is the bonafide work done by SHREE HARINI.K (9517202306092)(23BIT089),SOWMIYASHIVANI(9517202306096)(23BIT091),VARSHA.G.A (9517202306111) (23BIT096) for their Mini Project SMART PARKING LOT SYSTEM in the 19IT451 OPERATING SYSTEMS LABORATORY at Mepco Schlenk Engineering College,Sivakasi during the year2024-2025

SIGNATURE	**SIGNATURE**
Dr.N.MALATHY	**Dr.T.Revathi,M.E.,Ph.D.,**
STAFF IN-CHARGE	**HEADOFTHE DEPARTMENT**
Associate Professor	**SeniorProfessor**
InformationTechnology	**InformationTechnology**
Mepco SchlenkEngineeringCollege	**MepcoSchlenkEngineeringCollege**
VirudhunagarDt.–626005	**Virudhunagar Dt.–626005**

ACKNOWLEDGEMENT

We have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals and our reputed institution Mepco Schlenk Engineering College, Sivakasi.We would like to extend our sincere thanks to our Principal Dr. S. Arivazhagan ME., PhD.

We would like to express my special gratitude and thanks to Dr.T.Revathi ME..PhD.SeniorProfessor & Head of the Department for giving us such a wonderful experience.

We are highly indebted to Dr.N.MALATHY BE.,M.E.,PhD Associate Professor/IT, Mrs.K.Abirami BE., M.E,AP/IT for their guidance and constant supervision as well as for providing necessary information regarding the project & also for their support in completing the project.

We would like to express our gratitude towards other teaching and non teaching staff members of IT Department for their kind co-operation and encouragement which help us in all means to complete this project successfully.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

1.1 AIM

The primary aim of the Smart Parking Lot System is to develop an efficient and user-friendly solution for managing parking spaces in a way that minimizes the risk of deadlock and ensures optimal utilization of available resources. By implementing the Banker's Algorithm for deadlock avoidance, the system guarantees that no two vehicles are allocated the same parking space, thereby enhancing safety and operational efficiency. The project aims to generate a unique identification code for each vehicle upon entry, allowing for seamless tracking and management of parking slots. Additionally, the system will facilitate a straightforward process for users to vacate their parking space by entering the generated code, ensuring timely and accurate space reallocation. Ultimately, this project seeks to improve the overall parking experience for users while maximizing the efficiency of parking lot operations through advanced algorithms and hashing techniques.

1.2 OBJECTIVE

1. To implement a system that ensures each vehicle is assigned a unique parking space, preventing conflicts and maximizing space utilization.
2. To utilize the Banker's Algorithm to manage parking space allocation, ensuring that the system can avoid deadlock situations and maintain smooth operations.
3. To design an intuitive user interface that allows users to easily enter their vehicle information and receive a unique identification code upon parking.
4. To develop a mechanism for real-time tracking of occupied and available parking spaces, enabling efficient management of the parking lot.
5. To create a secure process for users to vacate their parking space by entering the unique code, ensuring that spaces are promptly made available for incoming vehicles.
6. To implement hashing techniques for secure storage and retrieval of vehicle information and unique codes, protecting user data from unauthorized access.

7. To establish metrics for monitoring system performance, including average parking time, space utilization rates, and user satisfaction, to continuously improve the system.
8. To design the system architecture in a way that allows for easy scalability, accommodating future expansions or integration with other smart city initiatives.
   Bookmark messageCopy messageExport

1.3.PROBLEM STATEMENT

In modern urban environments, managing parking spaces efficiently is a critical challenge due to increasing vehicle numbers and limited parking availability. Traditional parking systems often suffer from inefficient space allocation and lack dynamic resource management, leading to congestion, time wastage, and user frustration.

The objective of this project is to design and implement a Smart Parking Lot System that optimizes parking space allocation and management by leveraging hashing techniques to quickly identify and allocate available parking slots, and employing the Banker's algorithm to ensure deadlock-free and safe allocation of parking resources. This system aims to efficiently handle multiple concurrent parking requests, prevent resource allocation conflicts, and maximize the utilization of parking spaces while ensuring fairness and safety.

By integrating hashing and Banker's algorithm concepts from operating systems, the project seeks to overcome the limitations of conventional parking management systems and provide a scalable, responsive, and reliable solution suited for smart cities.

1.4.THE BANKER'S ALGORITHM

The Banker's Algorithm is a resource allocation and deadlock avoidance algorithm that is used in operating systems to manage resources among multiple processes. It was developed by Edsger Dijkstra and is named for its analogy to a banking system, where the bank must ensure that it can satisfy the needs of all its customers without running into a deadlock situation

1.5. KEY CONCEPTS

1. Resources and Processes: The algorithm deals with multiple processes that require resources to execute. Resources can be anything from CPU time, memory, I/O devices, etc.
2. Maximum Demand: Each process declares the maximum number of resources it may need. This is known as the maximum demand.
3. Current Allocation: The algorithm keeps track of how many resources are currently allocated to each process.
4. Available Resources: The algorithm maintains a count of the resources that are currently available for allocation.
5. Need Matrix: The algorithm calculates the remaining resources needed by each process, which is the difference between the maximum demand and the current allocation.

1.6. DEADLOCK

Deadlock is a situation in computer science and operating systems where two or more processes are unable to proceed because each is waiting for the other to release resources. In other words, a deadlock occurs when a set of processes are blocked because each process is holding a resource and waiting for another resource that is held by another process in the set. This results in a standstill where none of the processes can continue execution.

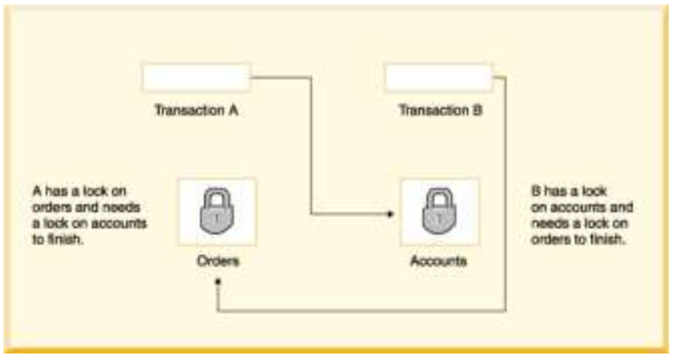

Transaction A
Transaction B
A has a lock on orders and needs a lock on accounts to finish.
Orders
Accounts
B has a lock on accounts and needs a lock on orders to finish.

FIG 1.6.1 DEADLOCK

1.7. CONDITIONS FOR DEADLOCK

For a deadlock to occur, the following four conditions must hold simultaneously:

Mutual Exclusion: At least one resource must be held in a non-shareable mode. That is, only one process can use the resource at any given time. If another process requests that resource, it must be delayed until the resource is released.
Hold and Wait: A process holding at least one resource is waiting to acquire additional resources that are currently being held by other processes. This means that processes can hold onto resources while waiting for others.
No Preemption: Resources cannot be forcibly taken from a process holding them. A resource can only be released voluntarily by the process holding it after it has completed its task.
Circular Wait: There exists a set of processes {P1, P2, ..., Pn} such that P1 is waiting for a resource held by P2, P2 is waiting for a resource held by P3, and so on, with Pn waiting for a resource held by P1, forming a circular chain.
Example of Deadlock
Consider a simple scenario with two processes (P1 and P2) and two resources (R1 and R2):
P1 holds R1 and is waiting for R2.
P2 holds R2 and is waiting for R1.
In this case, neither process can proceed because each is waiting for a resource that the other holds, resulting in a deadlock.

1.8.PREVENTION OF DEADLOCK

This involves ensuring that at least one of the four necessary conditions for deadlock cannot hold. For example
Mutual Exclusion can be avoided by making resources shareable.
Hold and Wait can be prevented by requiring processes to request all required resources at once.
No Preemption can be avoided by allowing resources to be forcibly taken from processes.

Circular Wait can be prevented by imposing a strict ordering of resource allocation.
Deadlock Avoidance: This involves using algorithms (like the Banker's Algorithm) to ensure that the system never enters an unsafe state. The system checks resource requests and only grants them if it can guarantee that the system will remain in a safe state.
Deadlock Detection and Recovery: In this approach, the system allows deadlocks to occur but has mechanisms to detect them and recover from them. This can involve terminating processes or preempting resources to break the deadlock.

CHAPTER 2
BANKER'S ALGORITHM

2.1. INTRODUCTION TO BANKER'S ALGORITHM

Banker's algorithm is a deadlock avoidance algorithm. The name was chosen because the algorithm could be used in a banking system to ensure that a bank never allocate its available cash in such a way that it could no longer satisfy the needs of a customers
When a new thread enters a system , it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in a system.when a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in safe state. If it will ,then the resources are allocated ,else the thread must wait until some other thread releases enough resources.

2.2. STATE DEFINITIONS

2.2.1. Safe State:

A state is considered safe if there exists a sequence of processes that can finish executing without causing a deadlock. In a safe state, the system can allocate resources to processes in such a way that all processes can eventually complete.

2.2.2 Unsafe State:

A state is unsafe if there is no guarantee that all processes can finish executing. This does not mean that a deadlock will occur, but it indicates that the system is at risk.

2.3. IMPLEMENTING BANKERS ALGORITHM

Several data structures are used to implement banker's algorithm.Here's a detailed explanation.
1. Available
It is an array of length m. It represents the number of available resources of each type. If

Available[j] = k, then there are k instances available, of resource type Rj.

2. Max
It is an n x m matrix which represents the maximum number of instances of each resource that a process can request. If Max[i][j] = k, then the process Pi can request atmost k instances of resource type Rj.

3. Allocation
It is an n x m matrix which represents the number of resources of each type currently allocated to each process. If Allocation[i][j] = k, then process Pi is currently allocated k instances of resource type Rj.

4. Need
It is a two-dimensional array. It is an n x m matrix which indicates the remaining resource needs of each process. If Need[i][j] = k, then process Pi may need k more instances of resource type Rj to complete its task.
Need[i][j] = Max[i][j] - Allocation [i][j]

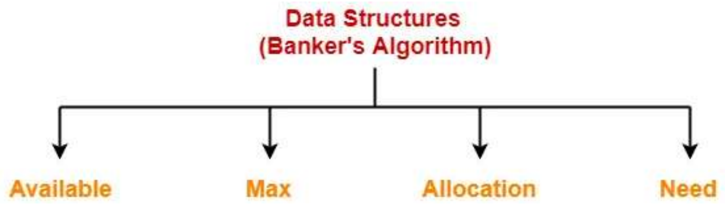


FIG 2.3.1

Banker’s Algorithm can be implemented using two different algorithms
(i)Safety algorithm
(ii)Resource Request Algorithm

2.3.1.SAFETY ALGORITHM

A safety algorithm is an algorithm used to find whether or not a system is in its safe state. The algorithm is as follows:

1. Let Work and Finish be vectors of length m and n, respectively. Initially,

Work = Available
Finish[i] =false for i = 0, 1, ... , n - 1.
This means, initially, no process has finished and the number of available resources is represented by the Available array.

2. Find an index i such that both

   Finish[i] ==false
   Need(i) <= Work
   If there is no such i present, then proceed to step 4.

It means, we need to find an unfinished process whose needs can be satisfied by the available resources. If no such process exists, just go to step 4.

3. Perform the following:

   Work = Work + Allocation(i)
   Finish[i] = true
   Go to step 2.

4. When an unfinished process is found, then the resources are allocated and the process is marked finished. And then, the loop is repeated to check the same for all other processes.
5. If Finish[i] == true for all i, then the system is in a safe state

That means if all processes are finished, then the system is in safe state.
This algorithm may require an order of mxn^2 operations in order to determine whether a state is safe or not.

2.3.2.RESOURCE REQUEST ALGORITHM

The resource-request algorithm and it is mainly used to determine whether requests can be safely granted or not.

Let Request(i) be the request vector for the process Pi. If Request i[j]==k, then process Pi wants k instance of Resource type Rj. When a request for resources is made by the process Pi, the following are the actions that will be taken:

1. If Request(i) <= Need(i), then go to step 2;else raise an error condition, since the process has exceeded its maximum claim.
2. If Request(i) <= Available(i) then go to step 3; else Pi must have to wait as resources are not available.
3. Now we will assume that resources are assigned to process Pi and thus perform the following steps:

Available = Available - Request(i);
Allocation(i) = Allocation(i) + Request(i);
Need(i) = Need(i) - Request(i);
If the resulting resource allocation state comes out to be safe, then the transaction is completed and, process Pi is allocated its resources. But in this case, if the new state is unsafe, then Pi waits for Request i, and the old resource-allocation state is restored

2.4. ALGORITHM & PSEUDOCODE FOR IMPLEMENTING BANKER'S ALGORITHM

2.4.1. ALGORITHM

- Define the number of processes (P) and resources (R).
- Initialize the Available, Max, Allocation, and Need matrices.
- Request Resources:
- When a process requests resources, check if the request is less than or equal to its Need and less than or equal to the Available resources.
- If the request is valid, temporarily allocate the resources and update the Available, Allocation, and Need matrices.

- Safety Check:
- After the temporary allocation, check if the system is in a safe state:
- Initialize a Work vector equal to Available and a Finish vector for all processes set to false.
- Find a process that can finish (i.e., its Need is less than or equal to Work).
- If such a process is found, simulate its completion by adding its Allocation to Work and marking it as finished.
- Repeat until all processes are finished or no more processes can be found.
- If all processes can finish, the system is in a safe state; otherwise, it is not.
- Final Decision:
- If the system is in a safe state after the request, grant the request.
- If not, roll back the temporary allocation and deny the request.

2.4.2.PSEUDOCODE

```
function request_resources(process_id, request):
    if request <= Need[process_id] and request <= Available:
        Available = Available - request
        Allocation[process_id] = Allocation[process_id] + request
        Need[process_id] = Need[process_id] - request
        if is_safe_state():
            return "Request granted"
        else:
            Available = Available + request
            Allocation[process_id] = Allocation[process_id] - request
            Need[process_id] = Need[process_id] + request
            return "Request denied"
    else:
        return "Request denied"


function is_safe_state():
    Work = Available
```

```
Finish = [false] * number_of_processes
while True:
    found = false
    for i in range(number_of_processes):
        if not Finish[i] and Need[i] <= Work:
            Work = Work + Allocation[i]
            Finish[i] = true
            found = true
    if not found:
        break
return all(Finish)
```

2.4. EXAMPLE

	allocation	max	available
Process	A B C	A B C	A B C
P0	1 1 2	4 3 3	2 1 0
P1	2 1 2	3 2 2	
P2	4 0 1	9 0 2	
P3	0 2 0	7 5 3	
P4	1 1 2	1 1 2	

1. The Content of the need matrix can be calculated by using the formula given below:

Need = Max – Allocation

Bankers Algorithm Table

2. Let us now check for the safe state.

Safe sequence:

For process P0, Need = (3, 2, 1) and

Available = (2, 1, 0)

Need <=Available = False

So, the system will move to the next process.

3.checking the process

For Process P1, Need = (1, 1, 0)

Available = (2, 1, 0)

Need <= Available = True

Request of P1 is granted

Available = Available +Allocation

= (2, 1, 0) + (2, 1, 2)

= (4, 2, 2) (New Available)

For Process P2, Need = (5, 0, 1

Available = (4, 2, 2)

Need <=Available = False

So, the system will move to the next process.

For Process P3, Need = (7, 3, 3)

Available = (4, 2, 2)

Need <=Available = False

So, the system will move to the next process

For Process P4, Need = (0, 0, 0)

Available = (4, 2, 2)

Need <= Available = True

Request of P4 is granted.

Available = Available + Allocation

= (4, 2, 2) + (1, 1, 2)

= (5, 3, 4)

Now again check for Process P2, Need = (5, 0, 1)

Available = (5, 3, 4)

Need <= Available = True

Request of P2 is granted

Available = Available + Allocation

= (5, 3, 4) + (4, 0, 1)
= (9, 3, 5) now, (New Available)

Now again check for Process P3, Need = (7, 3, 3)
Available = (9, 3, 5)
Need <=Available = True
The request for P3 is granted.
Available = Available +Allocation
= (9, 3, 5) + (0, 2, 0) = (9, 5, 5)

Now again check for Process P0, = Need (3, 2, 1)
= Available (9, 5, 5)
Need <= Available = True
So, the request will be granted to P0.
Safe sequence: < P1, P4, P2, P3, P0>
The system allocates all the needed resources to each process. So, we can say that the system is in a safe state.

The total amount of resources will be calculated by the following formula:
The total amount of resources= sum of columns of allocation + Available
= [8 5 7] + [2 1 0] = [10 6 7]

2.5. ADVANTAGES AND DISADVANTAGES

2.5.1.ADVANTAGES

- Deadlock Prevention: The primary advantage is its ability to prevent deadlocks by ensuring that resource allocation does not lead to an unsafe state.
- Dynamic Resource Management: The algorithm can handle dynamic changes in resource allocation and process demands, making it adaptable to varying workloads.

- Predictive Resource Allocation: By analyzing the maximum needs of processes, the algorithm can predict and manage resource allocation effectively.

2.5.2.DISADVANTAGES

- Overhead: The algorithm requires significant overhead for maintaining and checking the matrices, which can be computationally expensive, especially in systems with many processes and resources.
- Static Maximum Needs: The algorithm assumes that the maximum resource needs of processes are known in advance, which may not always be feasible in real-world applications.
- Limited Applicability: The Banker's Algorithm is best suited for systems with a fixed number of resources and processes. It may not be practical in highly dynamic environments where processes frequently enter and exit.

CHAPTER 3
IMPLEMENTATION TECHNIQUES

3.1. MODULES IDENTIFIED

1.Main Application Module

Class: Bankers

Responsibility: This is the entry point of the application. It initializes the main window and sets up the user interface.

Description: The constructor of the `Bankers` class sets the title, size, and default close operation for the JFrame. It also initializes the card layout for switching between different panels and sets up the parking panel, which contains all the UI components.

2. User Interface (UI) Module

Components: `JPanel`, `JButton`, `JTextField`, `JTextArea`, `JLabel`

Responsibility: Manages the graphical user interface elements and their layout.

Description: This module creates and arranges various UI components such as buttons for "Car In" and "Car Out", a text field for inputting car IDs, and a text area for displaying the parking status. It also sets the layout and background colors for the panels.

3. Resource Management Module

Variables: `available`, `max`, `allocation`, `need`, `availableCarIDs`, `carToSlot`, `usedCarIDs`

Responsibility: Tracks the resources available for parking, including the number of slots and the status of each slot.

Description: This module initializes arrays to manage the maximum resources, current allocations, and needs for each parking slot. It also maintains a queue of available car IDs and a mapping of car IDs to their respective slots.

4.Car Entry Handling Module

Method: `carIn()`

Responsibility: Handles the process of a car entering the parking lot.

Description: This synchronized method checks if there are available car IDs. If so, it generates a unique car ID, checks if the request can be safely granted using the Banker's Algorithm, finds an available slot, and updates the allocation and status accordingly. If the request cannot be granted, it shows an appropriate message.

5.Car Exit Handling Module

Method: `carOut(String carIDStr)`

Responsibility:Handles the process of a car exiting the parking lot.

Description: This synchronized method takes a car ID as input, checks if it is valid and currently parked, and then updates the allocation and available resources. It also updates the mapping of car IDs and shows a message indicating the successful removal of the car.

6.Banker's Algorithm Logic Module

Method: `isSafeState (int carIndex, int[] request)`

Responsibility:Implements the Banker's Algorithm to check if granting a resource request would leave the system in a safe state.

Description: This method checks if the requested resources exceed the car's needs or the available resources. It simulates the allocation and checks if all cars can finish their execution with the remaining resources. If a safe sequence exists, it returns true; otherwise, it returns false.

7.Slot Management Module

Method: `findAvailableSlot()`

Responsibility: Finds an available parking slot for a car.

Description: This method iterates through the slots to find one that is not currently allocated to any car. It returns the index of the first available slot or -1 if none are available.

8.Status Update Module

Method: `updateStatus()`

Responsibility: Updates the parking status displayed in the UI.

Description: This method constructs a string representation of the current parking status, including total slots, available slots, and parked cars. It updates the text area and the visual representation of the parking slots (colors and labels).

9.Message Display Module

Method:`showMessage(String message)`

Responsibility: Displays informational messages to the user.

Description: This method uses a JOptionPane to show messages, such as errors or confirmations, to the user in a dialog box.

10.Button Styling Module

Method: `createStyledButton(String text, Color bgColor)`

Responsibility: Creates and styles buttons for the UI.

Description: This method creates a JButton with specified text and background color, sets the font, foreground color, and other properties to ensure a consistent look and feel across the application.

11.Main Method

Method:`main(String[] args)

Responsibility: Entry point for the Java application.

Description: This method uses `SwingUtilities.invokeLater` to ensure that the GUI is created on the Event Dispatch Thread, which is the proper way to create and manipulate GUI components in Swing.

3.2. CONCEPT OF HASHING

3.2.1. HASHING

Hashing is a technique used in data structures to efficiently store and retrieve data. It involves transforming a given key (such as a number, string, or any other type of data) into an index in an array (called a hash table) using a hash function. This index is where the corresponding value will be stored or searched.

3.2.1.KEY CONCEPTS

1. HASH FUNCTION

- A function that takes a key as input and produces an integer, known as the hash code or hash value.
- The hash code is then mapped to an index in the hash table.
- Example: h(key) = key % table_size is a simple hash function where table_size is the size of the hash table.

2. HASH TABLE

- An array-like data structure where the hash values determine the index where the keys and their corresponding values are stored.
- Each index of the hash table is called a "bucket."

3. LOAD FACTOR

- The ratio of the number of elements in the hash table to the size of the hash table.
- It measures how full the hash table is and helps decide when to resize or rehash the table to reduce collisions.

3.2.3. EXAMPLES

Suppose we have a hash table of size 10 and we want to store values associated with keys

1. **Keys**: [5, 12, 15, 25]
2. **Hash Function**: h(key) = key % 10
3. **Mapping**
   - h(5) = 5 % 10 = 5 o h(12) = 12 % 10 = 2
   - h(15) = 15 % 10 = 5 (collision with 5)
   - h(25) = 25 % 10 = 5 (collision with 5 and 15)
4. **Visual Representation**:

Index: 0 1 2 3 4 5 6 7 8 9

Table: [ ] [ ] [12] [ ] [ ] [5 -> 15 -> 25] [ ] [ ] [ ] [ ]

Here, values 15 and 25 are chained to index 5 due to collisions. Using a good hash function and handling collisions effectively are crucial for efficient hashing.

3.2.4. Advantages of Hashing

• Fast Access: Typically provides constant time complexity, O(1), for search, insert, and delete operations, assuming good collision resolution

. • Efficient Memory Usage: Requires less memory compared to other data structures like arrays or linked lists for large data sets.

Use cases

• **Database Indexing**: Quickly locating records in databases.

• **Caching**: Storing frequently accessed data to reduce access time

• **Symbol Tables in Compilers**: Storing and quickly accessing variable and function names.

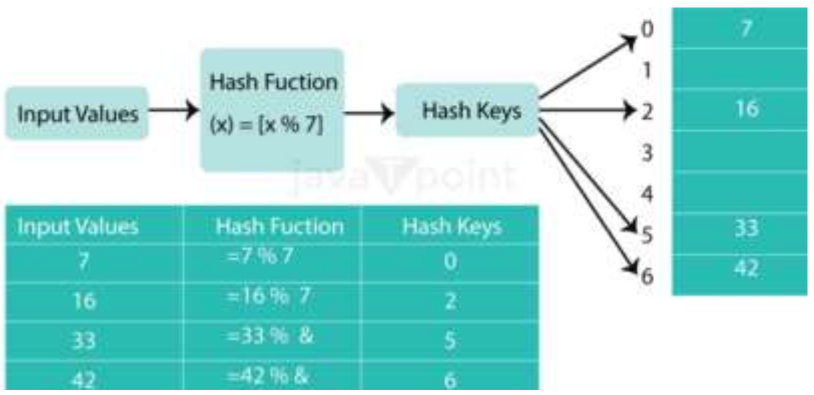
Hashing is a fundamental concept in computer science, used in various applications for its speed and efficiency in handling large datasets

3.2.5. HOW HASHING WORKS

The process of hashing can be broken down into three steps:

- **Input**:
  The data to be hashed is input into the hashing algorithm.
- **Hash Function**:
  The hashing algorithm takes the input data and applies a mathematical function to generate a fixed-size hash value. The hash value is returned, which is used as an index to store or retrieve data in a data structure.

3.3. ROLE OF HASHING IN OUR PROJECT



Input Values	Hash Fuction	Hash Keys
7	=7 % 7	0
16	=16 % 7	2
33	=33 % &	5
42	=42 % &	6

FIG 3.2.5.1

1. **Data Structure:**
   - The code uses a **HashMap<Integer, Integer>** called **carToSlot**. This map is used to associate each car ID (the key) with its corresponding parking slot number (the value).
2. **Hashing Mechanism**:

- When you add a key-value pair to a **HashMap**, Java internally uses a hash function to compute a hash code for the key (in this case, the car ID). This hash code is an integer that represents the key in a way that allows for efficient storage and retrieval.
- The hash function takes the car ID (an **Integer**) and computes its hash code using the **hashCode()** method provided by the **Integer** class. This method returns the integer value itself, as integers are already in a suitable format for hashing.

3. **Storage and Retrieval:**
    - The computed hash code is then used to determine the index in the underlying array where the key-value pair will be stored. This allows for average-case constant time complexity (O(1)) for both insertion and retrieval operations.
    - When you want to retrieve the value associated with a specific car ID, the same hash function is applied to the key to find the corresponding index in the array. If there are collisions (i.e., two different keys produce the same hash code), Java handles this using a technique called chaining, where multiple entries are stored in a linked list at the same index.
4. **Example Usage in the Code**:
    - The **carToSlot** map is used in several places in the code:
        - **Adding a Car**: When a car enters the parking lot, its ID is added to the 1carToSlot.put(carIndex, slotNumber);
        - **Removing a Car**: When a car exits, its ID is removed from the map. int slotNumber = carToSlot.remove(carID);
5. **Efficiency**:
    - The use of a **HashMap** allows for efficient lookups to check if a car ID is currently parked and to retrieve the associated slot number. This is crucial for the performance of the parking system, especially as the number of parked cars increases.

3.3.1 USE OF BANKER'S ALGORITHM

1. Data Structures Representing the Algorithm’s State

- available (int[]): Represents the number of available resources (parking slots).
  Initially, all slots are available: available[0] = TOTAL_SLOTS.
- max (int[][]): The maximum resource demand of each car.
  Each car's maximum is 1 slot (MAX_RESOURCES).
- allocation (int[][]): The currently allocated resources to each car (0 or 1 for each slot).
- need (int[][]): The remaining resource need for each car, initially equal to max (1 slot).

These represent the classical Banker's Algorithm matrices applied to this parking problem.

2. Key Method: isSafeState(int carIndex, int[] request)

This method performs the core Banker's Algorithm safety check for the given request.

Inputs:

- carIndex: the index (ID) of the car making the request for one parking slot.
- request: an array representing the number of resources requested (always [1] in this context, as a car requests exactly one slot).

Process:

It first ensures the request does not exceed the car’s total need or the currently available resources.

Using the Banker's Algorithm logic:

- **finish[]** array tracks which cars can finish (initially **false** for all).
- **work** array tracks available resources during simulation.
- The algorithm tries to find an order in which all cars can be allocated their needs from the simulated **work** resources.

It repeatedly checks if there is any unfinished car whose needs can be satisfied with current **work**. If yes:

- It marks that car as finished.
- Adds its allocated resources back to **work** (simulating it releasing resources).
- Updates the safe sequence list.

If, by the end, all cars can finish (safe sequence found), the method returns **true** indicating a safe state; otherwise, **false**.

3. Usage of the Banker's Algorithm in the Code

- When a car **enters** the system (**carIn()** method):
    1. It prepares a resource request of 1 slot.
    2. Calls **isSafeState(carIndex, request)** to verify if the system will stay safe after allocation.
    3. If safe, it allocates the slot and updates the state (**allocation**, **need**, **available**).
    4. If not safe, it denies the request and returns the car ID back to the available queue.
- When a car **leaves** (**carOut()** method), resources are released (allocation set to 0, need reset), and available slots increase, but no safety check is needed since releasing resources cannot cause unsafe states.

4. Initialization of Data Structures

Before the algorithm can be utilized, the necessary data structures must be initialized properly. This includes setting up the **available**, **max**, **allocation**, and **need** arrays.

5. Handling Resource Requests

When a car requests a parking slot, the system must ensure that the request is valid and does not lead to an unsafe state. The request handling process involves:

- **Validating the Request**: Before proceeding with the allocation, the system checks if the request exceeds the car's maximum need or the available resources.
- **Simulating the Allocation**: If the request is valid, the algorithm simulates the allocation by temporarily adjusting the **available**, **allocation**, and **need** arrays.

6. Safe State Verification

The core of the Banker's Algorithm is the safe state verification process. This involves:

- **Tracking Finished Cars**: The **finish** array keeps track of which cars have completed their parking process.
- **Work Array**: The **work** array simulates the available resources during the safety check.
- **Finding a Safe Sequence**: The algorithm iterates through the cars to find a sequence in which all cars can finish their parking without leading to deadlock.

7. Releasing Resources

When a car leaves the parking lot, the resources must be released properly. This involves:

- **Updating the Allocation**: The **allocation** for the car is set to zero, indicating that the slot is now free.
- **Updating the Need**: The **need** for that car is reset to its maximum demand.
- **Increasing Available Resources**: The **available** array is updated to reflect the newly freed slot.

The Banker's Algorithm is effectively integrated into the parking system to manage parking slot allocation safely. By simulating resource requests and ensuring that the system remains in a safe state, the algorithm prevents deadlocks and ensures efficient use of available resources. This implementation highlights the importance of resource management in systems where multiple entities compete for limited resources.

3.4. MODULE DESCRIPTION

1. Main Application Module

- Responsibility: Application entry point. Initializes the application frame and UI components.
- Description: This module creates the main JFrame, sets up the card layout for swapping screens, manages window properties, and starts the event dispatch thread.

2. User Interface (UI) Module

- Responsibility: Manages all graphical user interface components, layouts, and interactions.
- Description: Contains the construction of buttons (Car In, Car Out), text fields, labels, panels (parking lot, status area), coloring, and event listeners for user actions. Handles layout and styling to create a friendly and consistent UI.

3. Resource Management Module

- Responsibility: Tracks and manages resources like parking slots, available IDs, allocations, needs, and maximum resources.

- Description: Implements data structures such as arrays and queues for tracking available slots, car IDs, resource allocation matrices, and concurrency-safe operations on these.

4. **Banker's Algorithm Logic Module**
   - Responsibility: Implements the Banker's Algorithm safety check to ensure no unsafe state occurs when allocating parking slots.
   - Description: Contains the logic to verify whether resource requests can be safely granted without risk of deadlock in a multi-process/resource system (here adapted for parking allocation).

5. **Car Entry and Exit Handling Module**
   - Responsibility: Handles the car entering ("Car In") and leaving ("Car Out") actions, invoking resource checks and updating allocations.
   - Description: This module processes user commands for parking and unparking cars, allocates resources accordingly, updates internal mappings (car ID ↔ slot), and manages IDs in use.

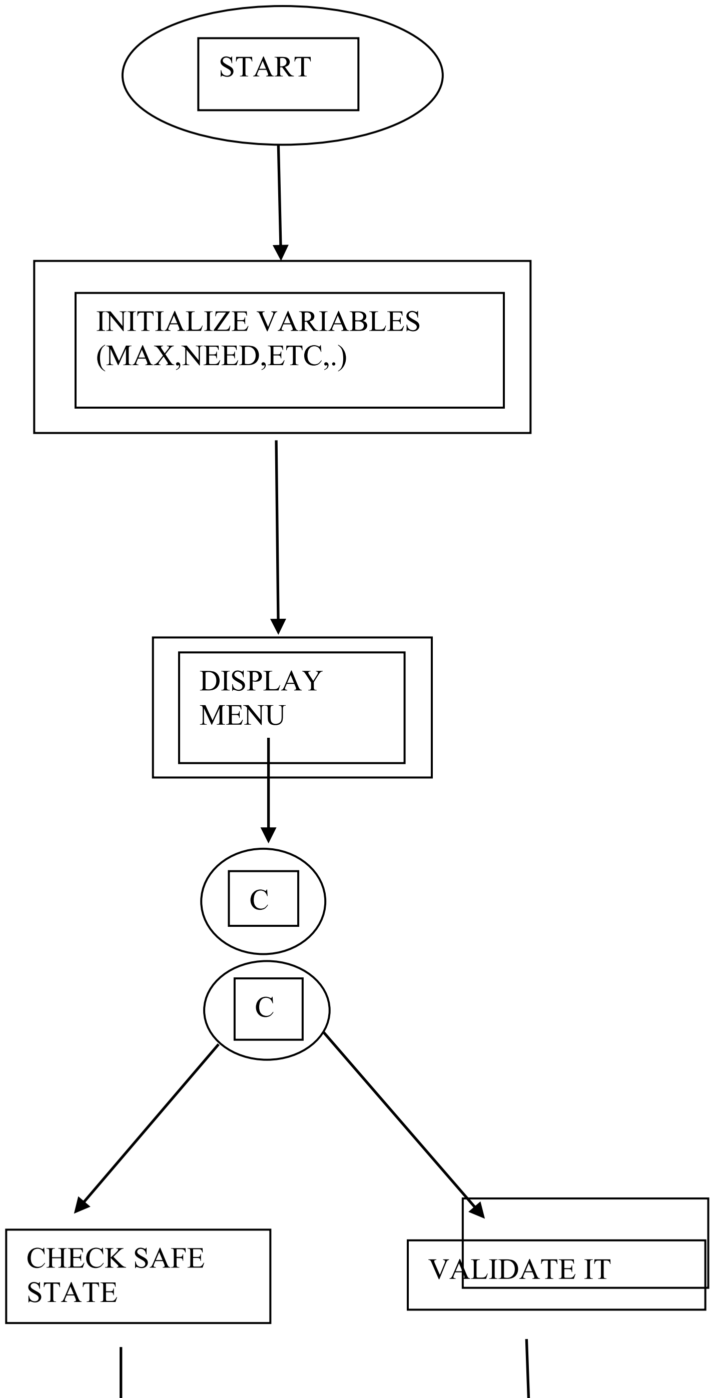
6. **Notification and Messaging Module**
   - Responsibility: Deals with displaying messages to users such as errors, successful allocations/deallocations, and system status.
   - Description: Uses dialog boxes and status text areas to provide relevant information and feedback to the user in an understandable manner.

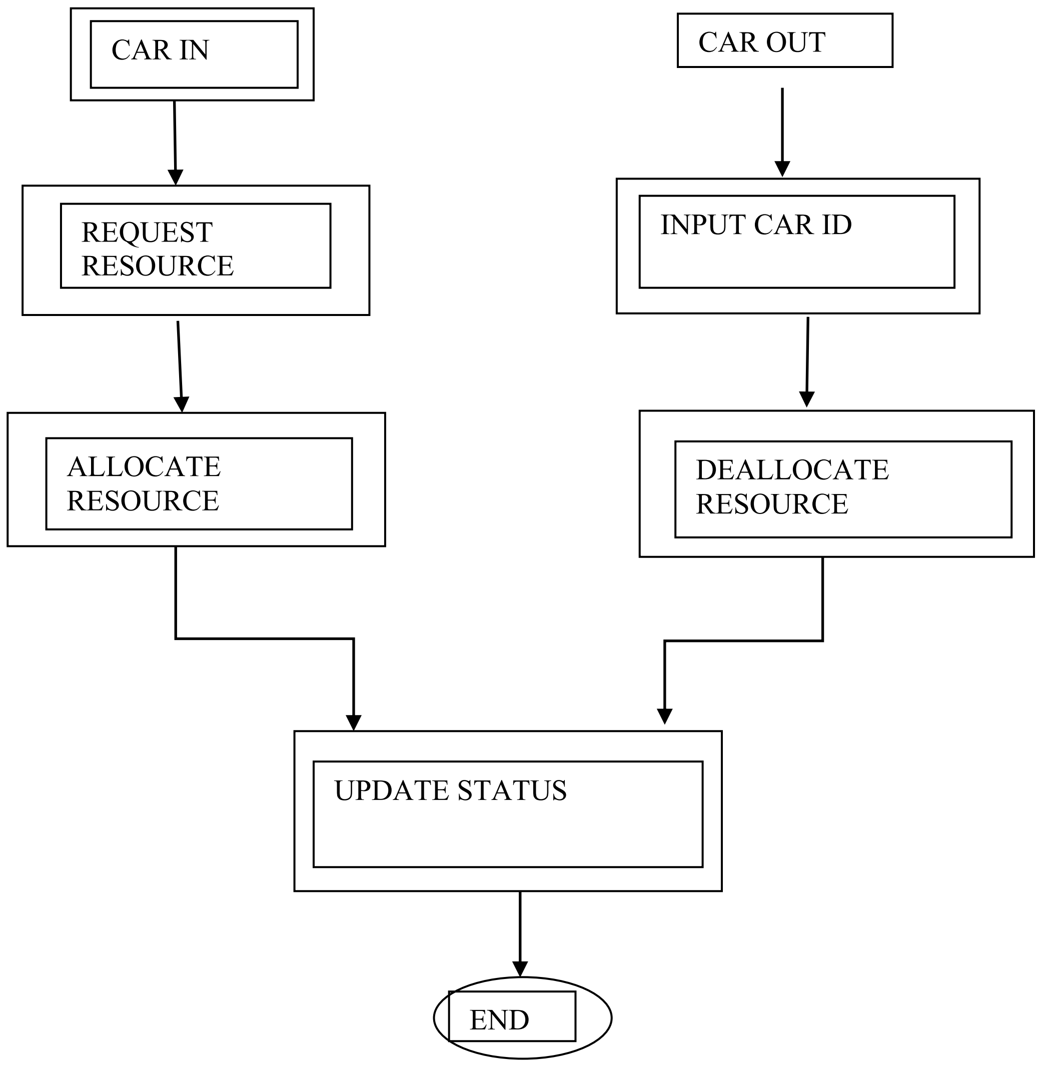
7. **Status Display and Update Module**
   - Responsibility: Updates the graphical representation of parking slots, colors, labels, and status text area to reflect the current parking lot state.
   - Description: Handles refreshing slot colors (green/red), car labels on slots, and the text area showing the summary of slot occupancy and car IDs.

8. Data Structures and Utility Module

- Responsibility: Provides utility methods and shared data structures used across other modules.
- Description: Could be extended with helper functions for data cloning, searching for slots, generating unique IDs, etc., ensuring clean separation of logic.

3.5. FLOW CHART





CHAPTER 4
PROGRAM

4.1.CODE

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.*;

public class Bankers extends JFrame {
    private static final int TOTAL_SLOTS =10;
    private static final int MAX_RESOURCES = 1;

    private int[] available;
    private int[][] max;
    private int[][] allocation;
    private int[][] need;

    private Queue<Integer> availableCarIDs = new LinkedList<>();
    private JTextArea statusArea;
    private JPanel[] slots = new JPanel[TOTAL_SLOTS];
    private JLabel[] carLabels = new JLabel[TOTAL_SLOTS];
    private Map<Integer, Integer> carToSlot = new HashMap<>();
    private Set<Integer> usedCarIDs = new HashSet<>();  // To track used IDs

    private CardLayout cardLayout;
    private JPanel mainPanel;

    public Bankers() {
        setTitle("Public Parking Lot - Banker's Algorithm");
        setSize(800, 600);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
```

```
cardLayout = new CardLayout();
mainPanel = new JPanel(cardLayout);

// Parking Screen
JPanel parkingPanel = new JPanel();
parkingPanel.setLayout(null);
parkingPanel.setBackground(new Color(220, 230, 241));

available = new int[]{TOTAL_SLOTS};
max = new int[TOTAL_SLOTS][1];
allocation = new int[TOTAL_SLOTS][1];
need = new int[TOTAL_SLOTS][1];

for (int i = 0; i < TOTAL_SLOTS; i++) {
   max[i][0] = MAX_RESOURCES;
   need[i][0] = MAX_RESOURCES;
   availableCarIDs.offer(i);
}

Font font = new Font("Segoe UI", Font.PLAIN, 16);

JLabel header = new JLabel("   Smart Parking System");
header.setFont(new Font("Segoe UI", Font.BOLD, 26));
header.setBounds(200, 10, 400, 40);
parkingPanel.add(header);

JButton enterBtn = createStyledButton("Car In", new Color(56, 142, 60));
enterBtn.setBounds(200, 70, 150, 40);
parkingPanel.add(enterBtn);

JButton exitBtn = createStyledButton("Car Out", new Color(211, 47, 47));
```

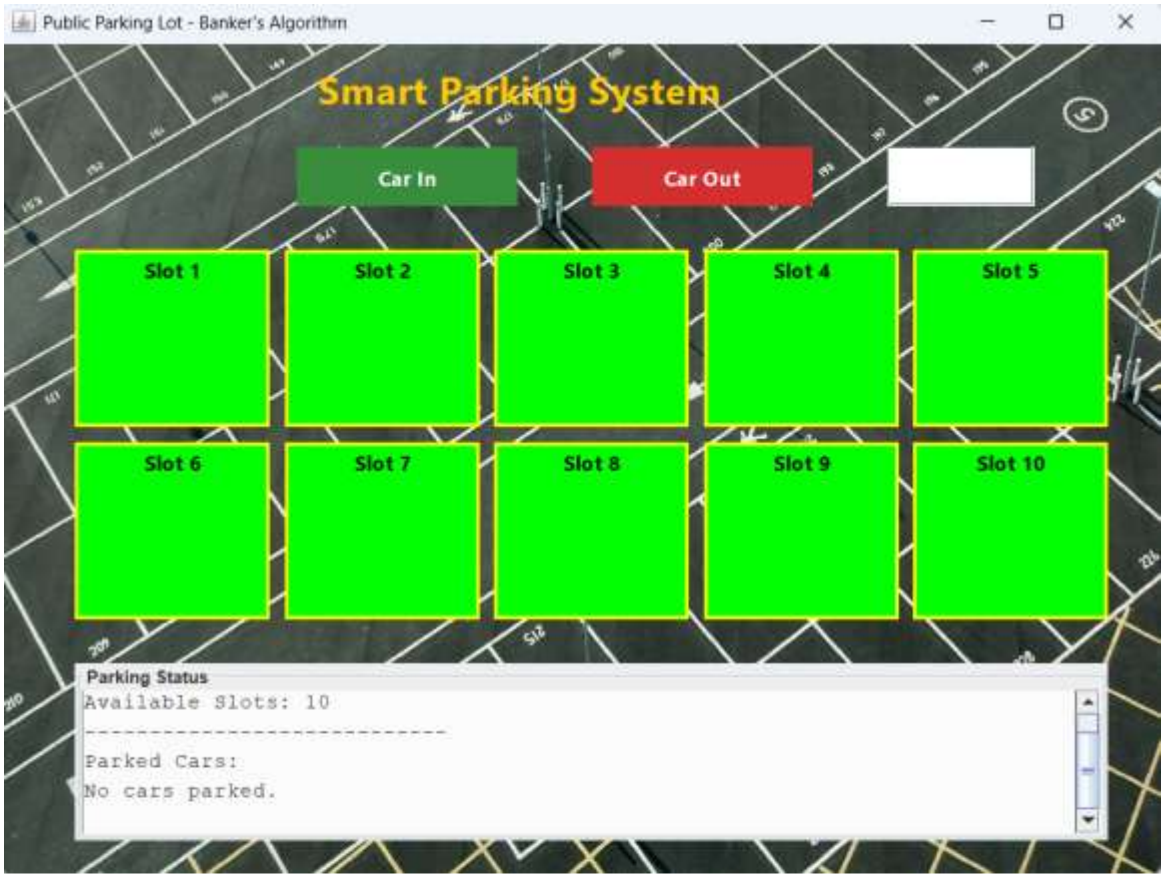
```
exitBtn.setBounds(400, 70, 150, 40);
parkingPanel.add(exitBtn);

JTextField exitField = new JTextField(10);
exitField.setBounds(600, 70, 100, 40);
parkingPanel.add(exitField);

statusArea = new JTextArea();
statusArea.setFont(new Font("Monospaced", Font.PLAIN, 14));
statusArea.setEditable(false);
statusArea.setBackground(new Color(245, 245, 245));
statusArea.setBorder(BorderFactory.createLineBorder(Color.GRAY, 1));
JScrollPane scrollPane = new JScrollPane(statusArea);
scrollPane.setBounds(50, 420, 700, 120);
scrollPane.setBorder(BorderFactory.createTitledBorder("Parking Status"));
parkingPanel.add(scrollPane);

// Custom Parking Lot Layout
JPanel lotPanel = new JPanel();
lotPanel.setBounds(50, 140, 700, 250);
lotPanel.setLayout(new GridLayout(2, 5, 10, 10));
lotPanel.setBackground(new Color(220, 230, 241));
parkingPanel.add(lotPanel);

for (int i = 0; i < TOTAL_SLOTS; i++) {
   slots[i] = new JPanel();
   slots[i].setBackground(Color.GREEN);
   slots[i].setBorder(BorderFactory.createTitledBorder("Slot " + (i + 1)));
   carLabels[i] = new JLabel();
   carLabels[i].setFont(new Font("Segoe UI", Font.BOLD, 14));
   carLabels[i].setForeground(Color.WHITE);
```

```
            slots[i].add(carLabels[i]);
            lotPanel.add(slots[i]);
        }

        enterBtn.addActionListener(e -> new Thread(this::carIn).start());
        exitBtn.addActionListener(e -> new Thread(() -> carOut(exitField.getText())).start());

        // Adding Panels to CardLayout
        mainPanel.add(parkingPanel, "Parking");
        add(mainPanel);

        updateStatus();
        setVisible(true);
    }

    private JButton createStyledButton(String text, Color bgColor) {
        JButton btn = new JButton(text);
        btn.setFont(new Font("Segoe UI", Font.BOLD, 14));
        btn.setForeground(Color.WHITE);
        btn.setBackground(bgColor);
        btn.setFocusPainted(false);
        btn.setBorder(BorderFactory.createEmptyBorder());
        btn.setCursor(new Cursor(Cursor.HAND_CURSOR));
        return btn;
    }

    private synchronized void carIn() {
        if (availableCarIDs.isEmpty()) {
            showMessage("No more car IDs available.");
            return;
        }
        int carIndex = availableCarIDs.poll();

        while (usedCarIDs.contains(carIndex)) {
            carIndex = availableCarIDs.poll();
        }

        int[] request = new int[]{1};

        if (isSafeState(carIndex, request)) {
            int slotNumber = findAvailableSlot();
            if (slotNumber != -1) {
                allocation[carIndex][0] = 1;
                need[carIndex][0] = 0;
                available[0]--;
                carToSlot.put(carIndex, slotNumber);
                usedCarIDs.add(carIndex);  // Mark the car ID as used
                updateStatus();
                displayUniqueCode(carIndex, slotNumber);
            } else {
                showMessage("No available slots.");
                availableCarIDs.offer(carIndex);
            }
        } else {
            showMessage("Request denied: Unsafe state.");
            availableCarIDs.offer(carIndex);
        }
    }

    private synchronized void carOut(String carIDStr) {
        try {
```

```
        int carID = Integer.parseInt(carIDStr);
        if (carToSlot.containsKey(carID) && allocation[carID][0] == 1) {
            allocation[carID][0] = 0;
            need[carID][0] = MAX_RESOURCES;
            available[0]++;
            int slotNumber = carToSlot.remove(carID);
            availableCarIDs.offer(carID);
            usedCarIDs.remove(carID);  // Remove the used car ID
            updateStatus();
            showMessage("Car " + carID + " removed from slot " + (slotNumber + 1));
        } else {
            showMessage("Invalid car ID.");
        }
    } catch (NumberFormatException e) {
        showMessage("Invalid input.");
    }
}

private boolean isSafeState(int carIndex, int[] request) {
    if (request[0] > need[carIndex][0] || request[0] > available[0]) {
        return false;
    }

    int[] tempAvailable = available.clone();
    int[][] tempAllocation = new int[TOTAL_SLOTS][1];
    int[][] tempNeed = new int[TOTAL_SLOTS][1];

    for (int i = 0; i < TOTAL_SLOTS; i++) {
        tempAllocation[i][0] = allocation[i][0];
        tempNeed[i][0] = need[i][0];
    }
    tempAvailable[0] -= request[0];
    tempAllocation[carIndex][0] += request[0];
    tempNeed[carIndex][0] -= request[0];

    boolean[] finish = new boolean[TOTAL_SLOTS];
    int[] work = tempAvailable.clone();
    ArrayList<Integer> safeSequence = new ArrayList<>();

    while (safeSequence.size() < TOTAL_SLOTS) {
        boolean found = false;
        for (int i = 0; i < TOTAL_SLOTS; i++) {
            if (!finish[i] && tempNeed[i][0] <= work[0]) {
                work[0] += tempAllocation[i][0];
                finish[i] = true;
                safeSequence.add(i);
                found = true;
            }
        }
        if (!found) return false;
    }

    return true;
}

private int findAvailableSlot() {
    for (int i = 0; i < TOTAL_SLOTS; i++) {
        boolean isAllocated = false;
        for (int j = 0; j < TOTAL_SLOTS; j++) {
            if (allocation[j][0] == 1 && carToSlot.get(j) == i) {
                isAllocated = true;
                break;
            }
        }
        if (!isAllocated) {
            return i;
        }
    }
    return -1;
}

private void updateStatus() {
    StringBuilder sb = new StringBuilder();
    sb.append("Total Slots: ").append(TOTAL_SLOTS).append("\n");
    sb.append("Available Slots: ").append(available[0]).append("\n");
    sb.append("----------------------------\n");
    sb.append("Parked Cars:\n");

    if (carToSlot.isEmpty()) {
        sb.append("No cars parked.\n");
    } else {
        for (Map.Entry<Integer, Integer> entry : carToSlot.entrySet()) {
            sb.append("vehicle ID: ").append(entry.getKey()).append(" in Slot: 
").append(entry.getValue() + 1).append("\n");
        }
    }

    statusArea.setText(sb.toString());

    for (int i = 0; i < TOTAL_SLOTS; i++) {
        slots[i].setBackground(Color.GREEN);
        carLabels[i].setText("");
    }

    for (Map.Entry<Integer, Integer> entry : carToSlot.entrySet()) {
        int slotNumber = entry.getValue();
        slots[slotNumber].setBackground(Color.RED);
        carLabels[slotNumber].setText("Car " + entry.getKey());
    }
}

private void displayUniqueCode(int carID, int slotNumber) {
    showMessage("vehicle ID: " + carID + " entered in slot " + (slotNumber + 1));
}

private void showMessage(String message) {
    JOptionPane.showMessageDialog(this, message, "Information",
JOptionPane.INFORMATION_MESSAGE);
}

public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> new Bankers());
}
```

CHAPTER 6
RESULTS

6.1. SCREEN SHOTS



```
Available Slots: 10
----------------------------
Parked Cars:
No cars parked.
```