**TITLE PAGE**

**DSA Project Documentation**

**Group Members:**

1. **Khadija Manzoor L1f23bsds0053**
2. **Hamza Jawad L1f23bsds0050**
3. **Muhammad Uzair Hanif L1f23bsds0054**

**Project Title: Parking Lot System**

**Problem Statement:**

Efficient management of parking spaces is a growing concern in urban areas where land is limited and demand for vehicle accommodation is high. In particular, small or single-lane parking facilities—such as alleyways, basement parking, or stack garages—face unique challenges where vehicles must be parked and removed in a strict sequence. These scenarios require that the most recently parked vehicle be removed first, making it impractical to retrieve a vehicle parked earlier without disturbing those parked afterward.

To deal with this problems ,this projects implements a Parking lot System based on Stack Data Structure which follows the Last-In, First-Out (LIFO) principles.

The system will provide the following core functionalities:

* Record vehicle entries and exits.
* Ensure proper stack-based handling of vehicle positioning.
* Display the current state of the parking lot to the user.

Here are some objectives:

**Objectives:**

* To design and implement a stack-based system that simulates LIFO vehicle entry and exit.
* To provide basic functionalities such as:
* Adding a vehicle
* Removing a vehicle
* Handling overflow (parking lot full) and underflow (no vehicle to remove) condition

Now we discuss some importance of this project why we use this:

**Importance:**

Efficient parking management is crucial for:

* Reducing traffic congestion in busy areas.
* Saving time during vehicle retrieval.
* Optimizing the use of limited parking spaces.
* Providing a practical demonstration of stack (LIFO) operations in real-life applications

**Tools and Technologies:**

**1)C++ Language:**

Here is some reasons why we used the C++ language:

**Reason of Selection:**

C++ is a powerful, high-performance, and widely-used programming language that provides low-level memory access and strong support for data structures like arrays and stacks. It allows direct control over memory, which makes it ideal for simulating stack operations efficiently.

Key Features we used:

* Linked List for implementing stack structure
* Conditional statements and loops for control flow
* Menu-driven interface using basic I/O

**2) Microsoft Visual Studio (IDE):**  
We used **Microsoft Visual Studio** as the integrated development environment (IDE) for writing, debugging, and compiling our C++ code.

**Benefits:**

* User-friendly and intuitive interface that simplifies coding and navigation.
* Powerful debugging tools, including breakpoints, watch windows, and step-through execution, which help quickly identify and fix issues.
* Seamless project and solution management, allowing easy organization of multiple files and dependencies.
* Integrated IntelliSense for code completion, syntax highlighting, and error detection, improving coding speed and accuracy.
* Comprehensive compiler support with optimized build configurations for efficient code compilation.
* Rich extensibility through plugins and extensions, allowing customization to fit specific development needs.
* Real-time code analysis and suggestions that enhance code quality and maintainability.
* Robust performance profiling and diagnostic tools to optimize application performance.

1. **OOP Concept:**

* CAR class represents each vehicle.
* Node class represents linked list nodes.
* Stack class encapsulates LIFO logic.
* ParkingLot class handles application-level functions like parking, exiting, and displaying cars.

**Advantages:**

* Improves code reusability and clarity.
* Separates concerns cleanly.

**Data Structures Used (with justification):**

Here are some data structures we used:

**1.Stack**

**Purpose:** The primary data structure used in this system is the Stack, implemented via a singly linked list.

**Why Stack?**

The Stack follows the **LIFO (Last In, First Out)** principle, which perfectly models the real-world behavior of a parking lot where:

* The **last car enters is the first onr to leave .**
* Cars can **only exists from one end**, just like in a single-lane parking area.

Here are some operation we used:

**Push():** Adds a new car to the parking lot.

**POP():** Removes the last parked car.

**Top():** Checks the car at the top .

**Size():** Returns the total number of parked cars.

**Empty():** Checks if the parking lot is empty.

#### ****Justification:****

* Efficient for managing **sequential entries and exits**.
* Avoids unnecessary complexity.
* Offers **O(1)** time complexity for insertion and deletion (push/pop).
* Mimics real-life limitations of tight parking lots.

**2. Singly Linked List**

#### ****Purpose:****

Each element in the stack is implemented as a **Node** that stores a Car object and a pointer to the next node.

#### ****Why Linked List?****

* **Dynamic memory allocation**: No need to predefine the maximum number of stack nodes (apart from logical capacity).
* Eliminates the limitation of fixed-size arrays.
* Allows easy creation and deletion of nodes at runtime.

#### ****Justification:****

* Flexible and scalable.
* Easily integrates with the stack class.
* Helps implement the stack without using built-in containers.

**3. Custom Classes (Abstraction & Encapsulation)**

#### ****Car Class:****

* Represents a vehicle using its **number plate**.
* Helps manage and identify individual cars in the system.

#### ****ParkingLot Class:****

* Controls stack operations.
* Encapsulates all user-facing features like ParkCar,exitCar,findCar etc.

#### ****Justification:****

* Follows OOP operations.
* Encourages **modularity and abstraction.**
* Makes future upgrades (like adding car models, timestamps) easy.

**Code Flow Explanation:**

1. **Program Initialization**
2. **File:** main.cpp
3. **Purpose:** Starts the program, sets parking lot capacity, and presents the main menu to the user.

**Menu-Driven User Interaction:**

* Inside the do while loop, a **menu** is displayed, allowing the user to:

**1. Park Car**

**2. Exit Car**

**3. Show Total Cars**

**4. Find Car**

**5. Show Parked Cars**

**6. Exit Program**

* Input is taken using cin
* **Validation is performed** to handle non-integer entries.

**Switch Case for User Choices:**

Each menu option corresponds to a function in the parkinglot class:

| **Option** | **Function** | **Description** |
| --- | --- | --- |
| 1 | parkCar() | Adds a car to the parking stack (if not full). |
| 2 | exitCar() | Removes the most recently parked car (top of stack). |
| 3 | totalCars() | Displays number of currently parked cars. |
| 4 | findCar() | Searches stack for a car by number. |
| 5 | showParkedCars() | Displays all parked cars from top to bottom. |
| 6 | Exit Program | Breaks the loop and ends the program. |

### **Stack Operations (LIFO):**

Implemented using a **linked list-based Stack** (Stack<T>), where each car is a Car object.

* push(Car) adds a new car to the top.
* pop() removes the top car (exit).
* top() accesses the car at the top.
* getTopNode() is used to traverse the stack in functions like findCar() and showParkedCars().

### **Object-Oriented Design Flow:**

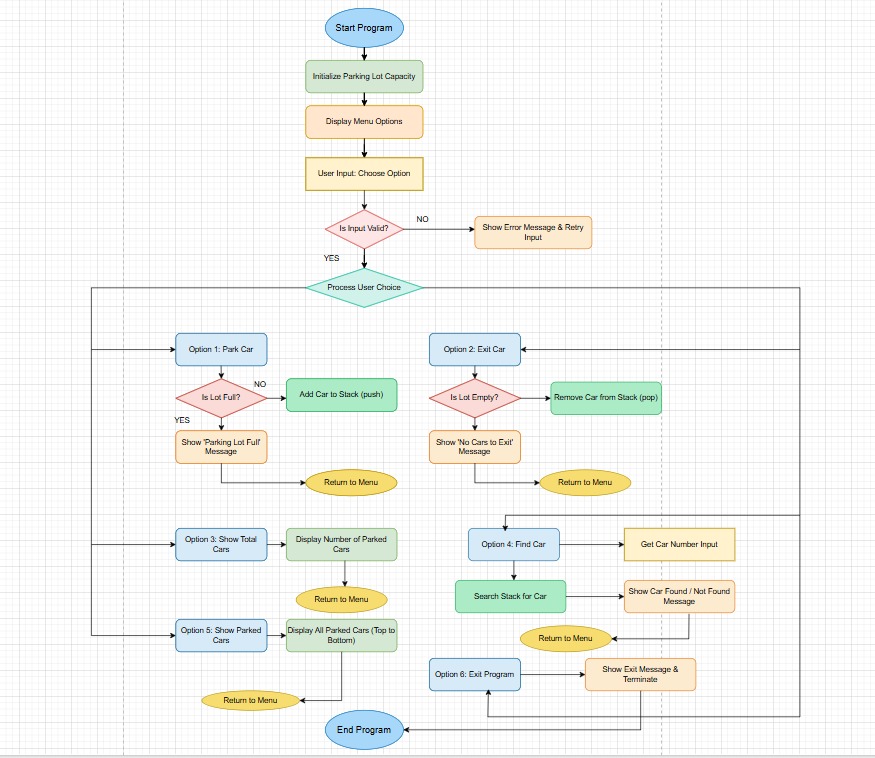
The program is divided into **modular classes**:

| **Class** | **Responsibility** |
| --- | --- |
| Car | Represents a car with a number. |
| Node<T> | Represents a node in the stack (template). |
| Stack<T> | Implements a stack using linked nodes. |
| ParkingLot | Controls parking operations, interfaces with the user. |

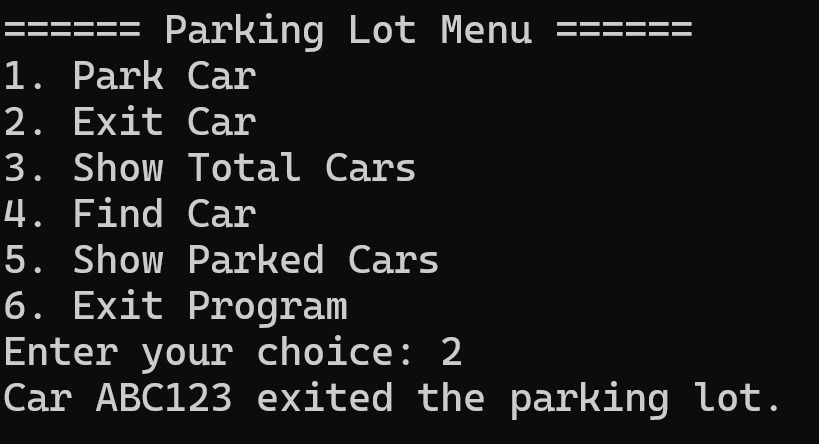
### **Program Termination:**

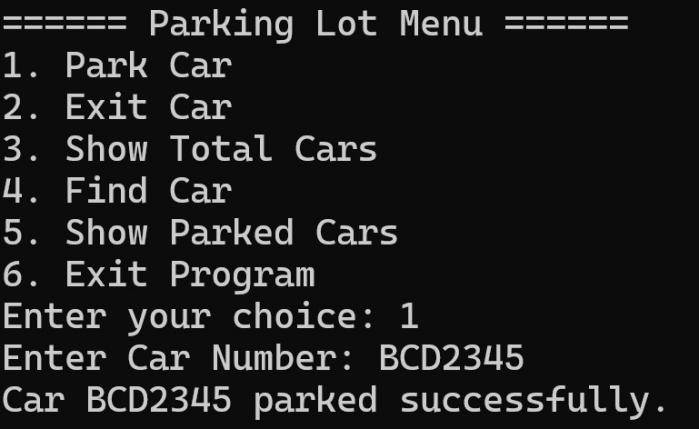
* When the user selects option **6**, the loop exits and the program terminates gracefully: cout << "Exiting program...\n";

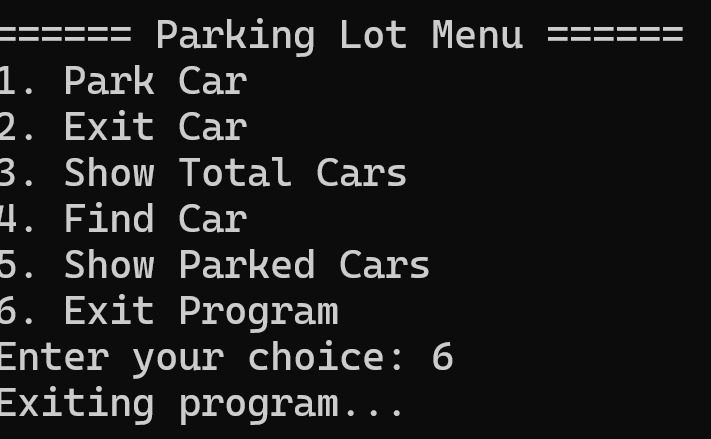
**Flowchart:**

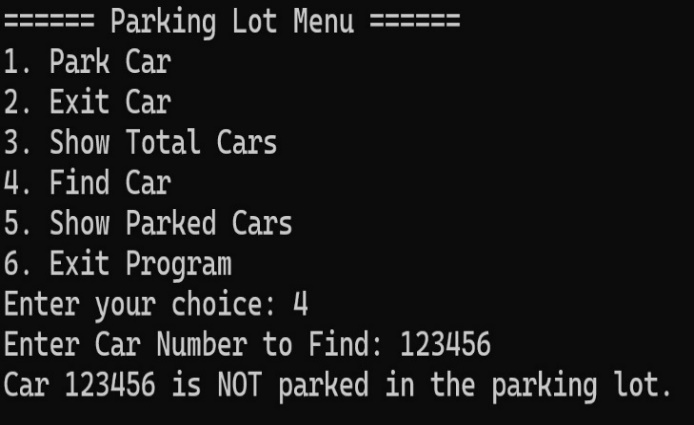
****

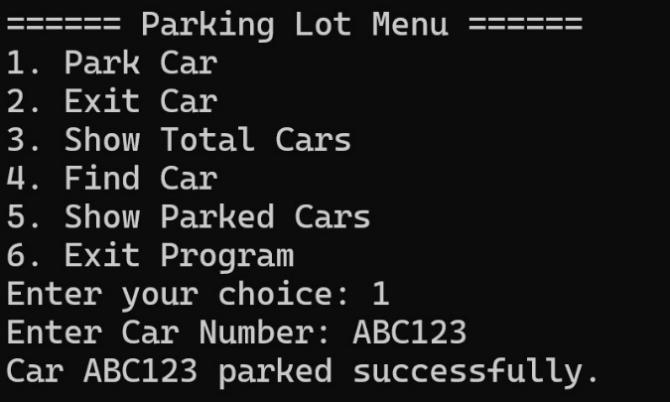
**Output Screenshots:**

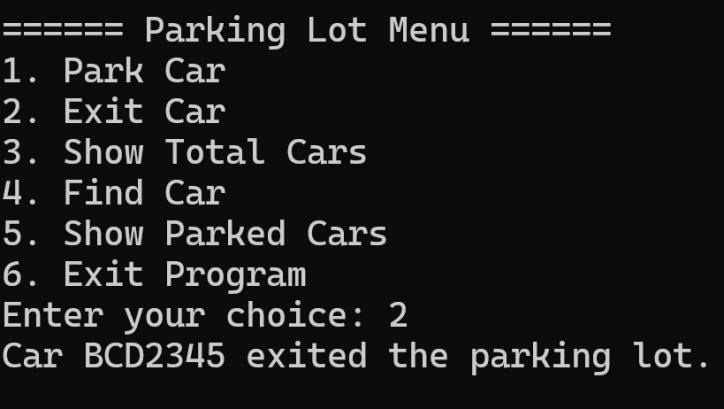
****

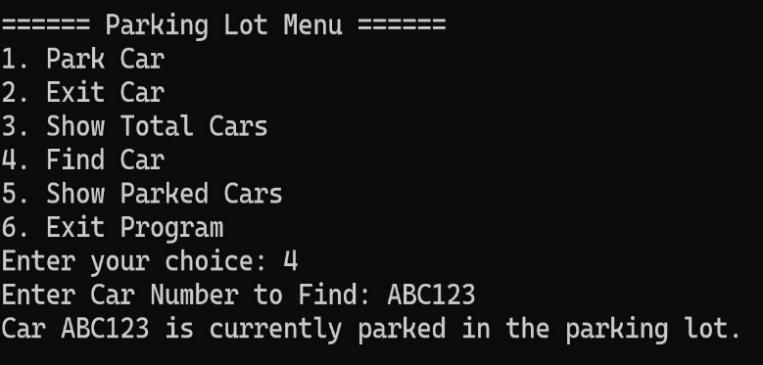
****

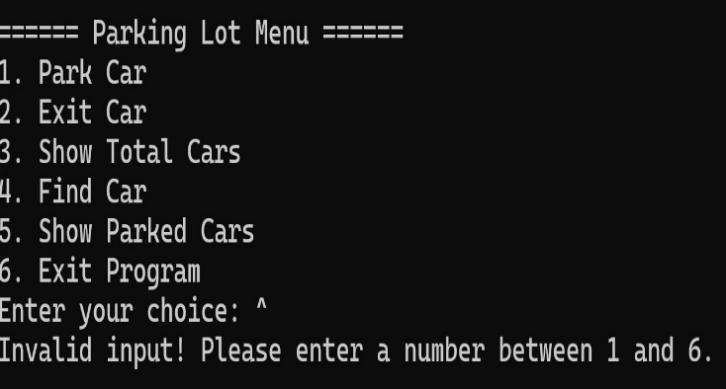
****

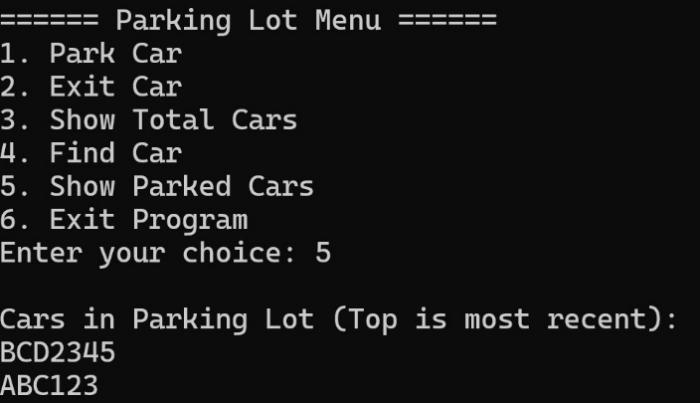


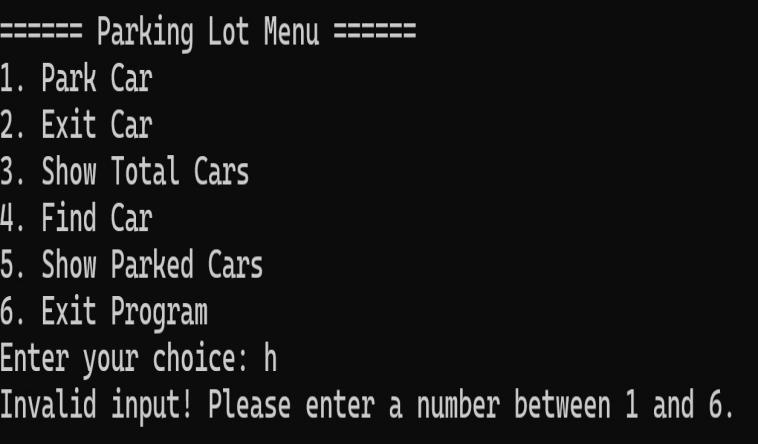


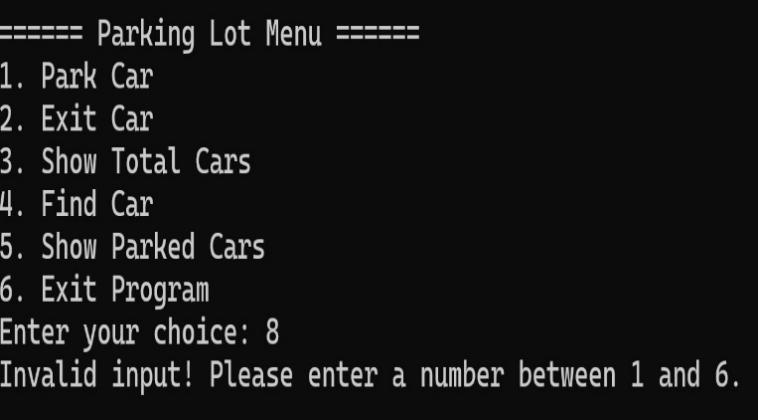


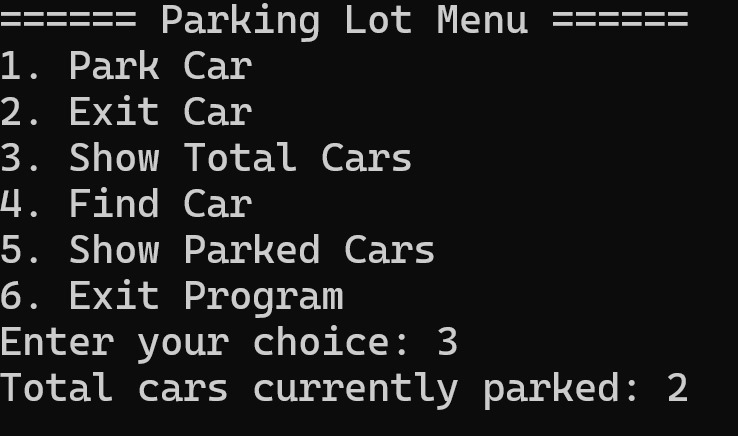






****

****

****

**Challenges and Conclusion:**

**Challenges:**

### 1. **LIFO Logic in Real-World Parking:**

* The Last-In-First-Out logic means the last car parked must leave first.
* In real scenarios, this is rarely practical — so simulating this purely with a stack required careful user interaction design and clearly defined assumptions.

### 2. **Dynamic Memory Managemen:**

* Implementing the stack using a **linked list** meant we had to manage memory manually using new and delete.
* Avoiding memory leaks and dangling pointers required precise control over Node creation and destruction.

### **3. Traversing the Stack (Non-Destructively):**

* For features like **searching** a car or **showing all parked cars**, we had to **traverse** the stack without modifying it.
* This was done safely using a pointer to the top node, but required care to ensure the original stack remained unchanged.

### 4. **Input Validation:**

* Handling invalid input from users (e.g., non-integer menu options) required additional code to clear and ignore incorrect streams.

### 5. **Capacity Constraints:**

* Since we implemented a fixed-size lot, it was important to constantly check whether the stack was full or empty before performing operations.

### 6**. Scalability Limitations:**

* The system works well for small-scale simulations, but would require significant redesign (e.g., queues or trees) to handle real-world complex parking systems.

**Conclusion:**

This project successfully simulates a **Parking Lot Management System** using a **Stack (LIFO)** data structure, implemented with a **linked list**. It provides essential features such as:

* Parking and exiting cars
* Viewing parked cars
* Finding a specific car
* Managing capacity and total cars

The system highlights how **data structures like stacks** can model real-world scenarios effectively under controlled assumptions. It also reinforces object-oriented programming principles like encapsulation, modularity, and class reuse.

While basic and educational in nature, this project forms a **foundation for more complex parking systems**, and can be extended with:

* Timed entry/exit
* Parking fees
* Real-time slot availability
* Multiple entry/exit lanes (requiring more advanced structures like queues or graphs)

This exercise enhances both programming logic and real-world problem-solving using C++.