Parking Management Project

Data Structures Course

Midterm Project

Prepared By:

Fatemeh Abshang

Amir Gerivani

Contents

1. Project Overview

This project focuses on modeling and managing a parking lot using fundamental data structures taught in the Data Structures course. The system simulates the real-world behavior of vehicles entering a parking facility, where car arrivals are managed through a **Queue**, and the parking spots inside the lot are represented by multiple independent **Stacks**, each implemented using a **Linked List**.

The parking lot supports a set of essential management operations, including:

- Enqueuing cars at the entrance.
- Parking a car into the first available stack or a user-selected stack.
- Searching for a car inside the lot.
- Removing a car only if it is at the top of its stack.
- Sorting individual stacks using the recursive Merge Sort algorithm.
- Transferring cars between stacks until capacity is reached.

The project integrates key concepts such as:

- Linked List implementation
- Stack and Queue using pointers
- Recursive algorithms
- Time complexity analysis
- GUI programming using Raylib
- Modular software architecture

This document explains the structure of the project, the responsibilities of each module, and provides an in-depth view of how the entire system is organized.

2. Project Structure and Components

The project is organized into a modular folder structure to ensure readability, maintainability, and logical separation of responsibilities. The main directory structure is as follows:

```
Parking_project/

 build.bat
 CMakeLists.txt
 README.md

 raylib/
    raylib-5.0_win64_mingw-w64/
    include/
    lib/

 src/
    main.cpp
    model/
    structures/
    parking/
    gui/

 bin/
     ParkingLot.exe
```

The core logic is located inside the `src/` directory. Below we examine each major component thoroughly.

2.1 model/ Directory

This directory contains the fundamental data models used across the system. These classes represent the minimal units of data such as Car and Node.

Its responsibilities include:

2.1.1 Car.h / Car.cpp

The `Car` class represents the fundamental data unit stored throughout the parking management system. Each car is modeled as an object containing essential identification information needed for search, comparison, and parking operations. The class is intentionally designed to remain lightweight and focused to ensure efficient storage within linked-list nodes.
The primary responsibilities of the `Car` class include:

- Storing a unique numerical identifier (`carId`) representing the vehicle.
- Encapsulating the identifier through proper getter and setter methods to maintain good object-oriented practices.
- Providing basic comparison functionality which enables linked-list based structures to search and locate specific vehicles.
- Acting as the payload stored inside each `Node` in both Stack and Queue implementations.

This abstraction allows the parking system to remain flexible and easily extendable. If additional metadata (such as entry time, owner information, or vehicle type) is needed in future extensions, it can be seamlessly integrated into the `Car` class without impacting the core logic of stacks, queues, or the parking manager.

2.1.2 Node.h / Node.cpp

The `Node` class is the fundamental building element of all linked-list based structures used in the project. Each node acts as a container for a single `Car` object and a pointer connecting it to the next node in the sequence. This modular design enables the creation of dynamic stacks and queues without relying on fixed-size arrays.
Every node consists of the following components:

- A `Car` instance representing the stored vehicle.
- A pointer to the next `Node` in the list, enabling traversal.

The `Node` class is intentionally minimalistic and does not include functionality beyond basic data storage and linkage. All logic for insertion, deletion, searching, and traversal is handled by higher-level structures such as `LinkedList`, `StackLL`, and `QueueLL`.
Nodes are essential in enabling:

- Dynamic allocation of cars without predefined memory limits.
- Efficient insertion and removal operations with time complexity $O(1)$ for stack and queue heads.
- Recursive algorithms such as merge sort for reorganizing stack contents.

Overall, the `Node` class forms the backbone of the data structures module and ensures that the entire parking system adheres strictly to the required linked-list–based implementation.

2.2 structures/ Directory

The `structures/` directory contains the fundamental data structures that form the backbone of the entire parking management system. All components inside this module are implemented manually using raw pointers, without relying on any pre-built libraries such as the C++ Standard Template Library (STL). This ensures full compliance with the project requirement mandating the explicit implementation of all linked-list, stack, and queue operations.

Each structure in this directory plays a key role in modeling realistic parking behaviors such as vehicle entry, exit, searching, sorting, and reallocation.

2.2.1 LinkedList.h / LinkedList.cpp

The `LinkedList` class provides a low-level, singly linked list implementation used as the foundation for both the stack and queue structures. This linked list manages memory dynamically and allows constant-time insertion and removal from the head of the list. Core functionalities of the `LinkedList` class include:

- **Node insertion and deletion:** Adding new nodes to the front of the list and removing nodes efficiently.
- **Traversal operations:** Iterating through nodes to access or inspect vehicle data as needed.
- **Search mechanisms:** Locating cars inside the list, which supports high-level features such as the `find()` function in the parking manager.
- **Utility functions:** Checking list emptiness, measuring size, and assisting other operations.

Although the user does not directly interact with the `LinkedList` class, it acts as an internal engine powering all higher-level abstractions. Its minimalistic design ensures fast operations, dynamic memory use, and smooth integration with recursive algorithms like merge sort.

2.2.2 StackLL.h / StackLL.cpp

The `StackLL` module implements a fully functional stack data structure using the `LinkedList` class. This component models each individual parking row, where cars are added and removed strictly based on Last-In-First-Out (LIFO) ordering — matching the real-world constraint that a car can exit only if no other cars block it.
Key operations provided by the `StackLL` class include:

- `push(Car)` – Inserts a car at the top of the stack.
- `pop()` – Removes the car at the top.
- `peek()` – Returns the car currently at the top without removing it.
- `isEmpty()` – Checks whether the stack contains any cars.
- `isFull()` – Ensures the stack does not exceed the parking capacity limit.

Beyond these fundamentals, the stack structure is essential for:

- Implementing realistic parking restrictions (cars in the middle cannot be removed).
- Supporting merge sort during the "sort a specific stack" operation.
- Assisting in the car relocation operation when transferring vehicles between stacks.

Overall, `StackLL` closely mirrors the behavior of real parking lanes and provides vital functionality for the system's physical simulation.

2.2.3 QueueLL.h / QueueLL.cpp

The `QueueLL` class models the entrance queue of the parking lot. This queue ensures that vehicles are admitted strictly in the order they arrive, thereby simulating first-come-first-served behavior. Implemented via a linked list, it provides efficient enqueue and dequeue operations at opposite ends of the structure.
The main operations of the `QueueLL` module include:

- `enqueue(Car)` – Adds a new incoming vehicle to the back of the queue.

- `dequeue()` – Removes and returns the car at the front.
- `front()` – Reads the next car to be admitted without removing it.

The entrance queue plays a vital role in maintaining:

- The real-world arrival order of vehicles.
- Predictable and fair distribution of cars into parking stacks.
- Proper behavior for operations such as "park in the first available stack."

Without this queue, vehicles would enter stacks unpredictably, breaking both project requirements and realistic parking logic.

2.3 parking/ Directory

The `parking/` directory contains the core operational logic of the entire system. Whereas the `structures/` module provides the data containers, the classes within this directory define how those structures interact to simulate real-world parking behavior. This includes parking cars, finding vehicles, sorting stacks, coordinating queue operations, and connecting system logic with the graphical user interface (GUI).

2.3.1 ParkingLot.h / ParkingLot.cpp

The `ParkingLot` class models the full physical layout of the parking area. It stores multiple stacks, each representing a lane in which cars park in a Last-In-First-Out manner. All high-level behaviors involving the structure and organization of parked cars are implemented here.
The class maintains the following essential components:

- An array of `StackLL` objects, each representing a single parking row.
- A fixed capacity for each stack, enforcing the maximum number of cars allowed per row.
- The total number of stacks available in the parking lot.

Key functionalities implemented in the `ParkingLot` class include:

- **Parking cars in the first available stack:** Removes a vehicle from the queue and places it into the earliest non-full stack.
- **Parking into a user-selected stack:** Allows the GUI or user input to specify the destination stack.

- **Removing cars from the top of a stack:** Enforces the rule that only the topmost vehicle in any stack may exit the parking lot.
- **Finding a specific vehicle:** Traverses every stack and every node to determine the exact location of a specified car.
- **Merge-sorting an individual stack:** Applies a recursive merge sort algorithm directly on the underlying linked list of that stack.
- **Transferring cars between stacks:** Moves vehicles from one stack to another and automatically spills excess cars into subsequent stacks if the destination becomes full.

This class effectively models the physical constraints of a real parking lot, providing the underlying implementation for all operational rules and ensuring safety, consistency, and correctness of behavior across the system.

2.3.2 ParkingManager.h / ParkingManager.cpp

The `ParkingManager` class serves as the high-level controller of the application. It acts as the central coordinator that links user interactions from the GUI to the underlying parking logic contained in the `ParkingLot`. While `ParkingLot` focuses on data-level operations, `ParkingManager` focuses on workflow, validation, and communication.
Primary responsibilities of the `ParkingManager` include:

- Managing the entrance queue of arriving cars through the `QueueLL` structure.
- Receiving commands and events from the graphical interface and translating them into parking actions.
- Validating user input (such as stack selection or car ID) before forwarding it to the parking lot.
- Coordinating complex operations — for example, moving cars across multiple stacks or triggering a stack sort.
- Updating the GUI with the latest system state after each operation.

Conceptually, this class functions as the "brain" of the entire project. It ensures consistent behavior between all components, prevents illegal operations, and maintains system stability. The `ParkingManager` abstracts the internal complexity of the data structures and exposes a clean set of high-level functions for GUI interaction.

Whereas the `structures/` module provides the data containers, the classes within this directory define how those structures interact to simulate real-world parking behavior.

This includes parking cars, finding vehicles, sorting stacks, coordinating queue operations, and connecting system logic with the graphical user interface (GUI).

2.4 gui/ Directory

The Graphical User Interface (GUI) of this project is implemented using **Raylib**, a lightweight and efficient graphics framework designed for real-time visualization and interactive applications. This module is responsible for displaying the parking system on the screen and providing an intuitive way for users to interact with the program. The GUI acts as the visual representation layer on top of the underlying data structures and logic.

2.4.1 RaylibUtils.h / RaylibUtils.cpp

This component contains a collection of rendering and utility functions that abstract away the low-level Raylib drawing calls. Its primary role is to visually represent the state of the parking lot and its associated structures at every frame.
Key responsibilities include:

- **Rendering the parking lot visually:** Conversion of logical data (cars, stacks, queues) into graphical objects on the screen.
- **Drawing linked-list-based stacks and queues:** Each stack is represented as a vertical column of car blocks, while the queue is drawn horizontally. The module ensures correct spacing, alignment, and visibility.
- **Displaying individual cars:** Cars are drawn using color-coded rectangles or textures, and each car's unique identifier (`carId`) is rendered on top for clarity.
- **Rendering interactive UI components:** Buttons, labels, backgrounds, and highlights are drawn here. These visual elements allow the user to perform operations such as adding cars, removing cars, or sorting stacks.
- **Managing layout and dynamic updates:** Whenever a change occurs in the underlying logic—such as a parked car, a removed car, or a rearranged stack—the module updates the visuals immediately to reflect the current system state.
- **Providing reusable drawing helpers:** This includes functions for drawing rectangles, text, arrows, list diagrams, and other graphic primitives that are repeatedly used across the interface.

2.4.2 InputHandler.h / InputHandler.cpp

This module is responsible for processing all user-driven input events and translating them into actions that affect the parking system. It acts as the communication layer between the GUI and the logical engine (`ParkingManager`).
Its main responsibilities include:

- **Handling mouse interactions:** Detecting left and right clicks, determining whether the user clicked on a button, a specific stack, or empty space, and triggering the corresponding action.
- **Processing keyboard input:** Supporting keyboard shortcuts or numeric input (e.g., entering a car ID manually), depending on the features enabled in the GUI.
- **UI event detection and routing:** Checking if the user selected operations such as:
  - Park the next car in the queue
  - Remove a car from a specific stack
  - Sort a stack visually
  - Move cars between stacks
  - Load or clear the parking lot

  Once detected, these events are forwarded to the logical core.
- **Triggering actions through the `ParkingManager`:** The module acts strictly as an input interpreter; it does not modify car data directly. Instead, it invokes the appropriate methods in `ParkingManager`, ensuring a clean separation between user interactions and system logic.
- **Providing feedback for invalid actions:** For example, clicking on a full stack or attempting an unsupported action. Visual cues such as highlighting buttons or printing warning messages can be triggered here.

Together, the `RaylibUtils` and `InputHandler` modules create a responsive, intuitive, and visually clear interface that allows users to interact with a system built on linked lists, stacks, and queues—all in real time.

2.5 main.cpp

The `main.cpp` file represents the central entry point of the application and integrates all components of the system into a fully functioning interactive program. This file is

responsible for initializing the graphical environment, constructing the core logic modules, and maintaining the continuous render loop required for real-time visualization and user interaction.
The major responsibilities of `main.cpp` include:

- **Initializing the Raylib graphics environment:** Setting up the window size, title, frame rate, and preparing the rendering context that will be used throughout the program. This initialization ensures that graphical functions inside the GUI module can operate smoothly.
- **Constructing core system components:** `main.cpp` creates and initializes:
  - the `ParkingManager` (central controller),
  - the `ParkingLot` (which holds all stacks),
  - the entrance queue,
  - GUI helper modules such as `RaylibUtils` and `InputHandler`.

  These objects are interconnected and form the complete operational environment of the program.
- **Running the main application loop:** Raylib applications generally follow a continuous loop structure. Inside this loop, the program:
  - processes user inputs via the `InputHandler`,
  - updates system state by invoking methods on `ParkingManager`,
  - renders the current parking lot, queue, and UI elements through `RaylibUtils`.

  This loop keeps running until the user closes the application window.
- **Ensuring smooth synchrony between logic and graphics:** `main.cpp` acts as the bridge between the backend logic and the GUI layer. It ensures that after every operation—such as parking a car, removing one, sorting a stack, or relocating vehicles—the visual representation is updated instantly.
- **Graceful shutdown and cleanup:** Once the user exits the application, `main.cpp` is responsible for terminating the Raylib window, releasing resources, and performing any end-of-program actions needed for a clean exit.

In essence, `mai` orchestrates all modules and ensures that the program behaves as an integrated, interactive system. Without this file, the individual components of the project would remain isolated and unable to form a coherent application.

3. Conclusion

Throughout this project, we developed a fully functional parking management system integrating core data structures such as queues, stacks, and linked lists. The system includes advanced features such as recursive merge-sort, stack-to-stack transfers, and car search functionalities.

In addition, the project includes:

- A complete GitHub repository with meaningful commits.
- A graphical user interface (GUI) implemented using Raylib.
- A modular architecture with clear separation of responsibilities.

This project provided valuable experience in implementing data structures manually, designing modular C++ software, and building an interactive UI, all aligned with real-world engineering principles.