Hotel California Elevator Management System

Student: Saleh zabihipazhouh &ID: 40334184
Course: Data Structures • **Language:** Python & C

Abstract

This project implements a simulation of a smart elevator management system for "Hotel California," a building with 41 floors (-10 to 30). The system is designed to efficiently handle user requests using a dual-elevator setup. The implementation includes: (1) A **Python prototype** used for high-level logic verification and simulation of real-time physics; and (2) A **C implementation** demonstrating manual memory management (malloc/free), pointer manipulation, and structural organization. The core algorithm utilizes **Sorted Linked Lists** to implement a modified **SCAN algorithm** (Elevator Algorithm), ensuring requests are served in an optimal order. It features a custom **Cost Function** for dispatching, **Priority Queues** for VIP handling, and an **Emergency Mode**. This hybrid approach ensures modularity, scalability, and adherence to strict data structure requirements.

Table of Contents

1. Project Requirements.....2
2. Problem Definition & Background.....2
3. Data Structures Used.....3
4. Algorithmic Approach.....3
   4.1 Priority Queues via Sorted Linked Lists.....3
   4.2 The Cost-Based Dispatching Algorithm.....4
   4.3 VIP & Absolute Priority Handling.....5
5. Implementation Architecture.....6
6. Code Structure & Modules.....7
7. Complexity Analysis.....8
8. Testing & Validation.....8
9. Design Decisions (Python vs C) 9

---

1. Project Requirements

- **Building Constraints:** 41 Floors (Range: -10 to +30). Two Elevators (A and B).
- **Data Structures:** Must demonstrate understanding of **Queues**, **Stacks**, **Arrays**, and **Linked Lists**.
- **Priority System:** Implement a logic to prioritize requests based on proximity and direction.
- **Simulation:** Clear output showing elevator movement (Moving Up, Moving Down, Door Opening).
- **Language Conversion:** Logic must be implemented in Python first, then converted to C to demonstrate memory management.
- **Bonus Features:** Implementation of **Emergency Mode** (stop and clear) and **VIP Priority** (absolute priority over normal users).

2. Problem Definition & Background

The Elevator Problem (or Disk Scheduling Problem) involves scheduling a resource (elevator) to service a set of requests (floors) to minimize total travel time and wait time. A naive First-Come-First-Served (FCFS) approach leads to "zig-zag" movement and inefficiency.
This project implements the **SCAN Algorithm** (Elevator Algorithm), where the elevator continues in one direction until all requests in that direction are satisfied before reversing.

Key constraints handled:

- **Directional Physics:** An elevator moving UP cannot instantly reverse to serve a lower floor without stopping.
- **Target Floor (tf): The system tracks the furthest destination to calculate turn-around costs.**

3. Data Structures Used

The project relies on specific structures to manage state:

1. **Sorted Linked List (Priority Queue):** Used to store pending requests.
    - *Why?* Allows O(N) insertion in sorted order. This automatically arranges requests (e.g., 5, 7, 10) so the elevator visits them sequentially without sorting an array every step.

2. **Stack (LIFO):** Used to store the history of visited floors.
    - *Why?* The most recent floor visited is the most relevant for history tracking.
3. **Struct/Class (Elevator):** Encapsulates state (current_floor, direction, target_floor, state).

---

4. Algorithmic Approach

4.1 Priority Queues via Sorted Linked Lists

Instead of a standard queue, we use a sorted linked list.

- **UP Queue:** Sorted Ascending (−2→5→10 \to 5 \to 10→5→-2).
- **DOWN Queue:** Sorted Descending (10→5→−210 \to 5 \to -210→5→−2).

4.2 The Cost-Based Dispatching Algorithm

To decide which elevator (A or B) gets a request, we calculate a "Cost" score. Lower cost wins.

The Formula:

1. **Idle:** $Cost = |Current - Request|$
2. **Moving Towards:** $Cost = |Current - Request|$
3. **Moving Away:** $Cost = |Current - Target| + |Target - Request|$

This ensures that if Elevator A is at floor 0 going to 30, and a request comes for floor -2, it is penalized heavily because it must travel all the way to 30 and back.

Complexity: O(1) constant time math calculation.

4.3 VIP & Absolute Priority Handling

VIP requests are stored in separate queues (vip_up, vip_down).
The movement logic grants **Absolute Priority**:

1. If a VIP exists in the current direction, serve them.
2. If a VIP exists in the *opposite* direction, **switch direction immediately**, ignoring normal requests in the current direction.

3. Only serve normal queues if no VIPs exist.

---

5. Implementation Architecture

The system is designed with a modular approach:

- **Node Module:** Defines the structure for Linked List nodes.
- **Stack Module:** Manages the history array with a fixed size (10) and shift logic.
- **Elevator Module:** Contains the core state machine (IDLE, UP, DOWN, EMERGENCY) and the four queues (up, down, vip_up, vip_down).
- **Controller (HotelSystem):** Receives requests and dispatches them to the best elevator based on calc_cost().

6. Main Program & Validation

The main function simulates a time-based series of events to demonstrate the logic:

1. **Initialization:** Elevators A and B are instantiated at different floors.
2. **Scenario Injection:**
    - Requests are added (e.g., Floor 5, Floor 25).
    - Conflict scenarios are tested (e.g., Requesting floor -2 when A is moving UP).
    - VIP priority is tested (Requesting 10 with vip=True).
3. **Simulation Loop:** The move_step() function is called in a loop to simulate the passage of time (physics), printing ^ (Up), v (Down), and * DING * (Arrival).

---

7. Complexity & Guarantees

- **Insertion (Add Request):** O(N)
    - Because we use a Sorted Linked List, we must traverse the list to find the correct insertion spot. Since *N floors) is small (41), this is negligible*.
- **Dispatching (Cost Calculation):** O(1)
    - Mathematical calculation involving only current state variables.

- **Movement Step:** O(1)
    - Checking the head of the queues and incrementing/decrementing floor counters.
- **Space Complexity:** O(N)
    - Proportional to the number of active requests stored in nodes.

8. Testing & Validation

- **Memory Safety (C):** Checked for memory leaks. The emergency() function manually iterates through all linked lists and calls free() on every node.
- **Duplicate Handling:** Logic ensures that if a user requests floor 5 multiple times, only one node is created.
- **Boundary Checks:** Requests below -10 or above 30 are rejected.
- **Emergency Reset:** Validated that the system correctly locks during emergency and clears all queues, then resets to IDLE upon manual reset.

9. Design Decisions & Alternatives

- **Why Linked Lists over Arrays?**
    - Arrays require shifting elements when inserting in the middle (for priority sorting). Linked lists allow O(1) pointer changes once the location is found.
- **Why C and Python?**
    - Python allowed for rapid prototyping of the complex "Cost" and "VIP" logic.
    - C demonstrated low-level understanding of how data structures are stored in memory (Stack vs Heap).
- **Target Floor (**tf**) Strategy:**
    - We track the "Target Floor" (tf) dynamically. It represents the furthest point the elevator is committed to. This prevents the cost function from fluctuating wildly when the elevator is mid-journey.
