Program Structure and Development

Mingzheng Michael Huang

Translating Algorithm Components

- Repetition: Utilized while loops extensively to traverse and interact with the singly-linked list. For instance, in the Add(), Remove(), Print(), and various Count*() functions.
- Decision Making: Employed if/else statements for several purposes: Input validation in Add(), Comparing flight data, such as flightnumber or time, in functions like CountSmaller() and CountLater().
- Data Handling: Structs (Flightinfo and Item) were defined to encapsulate flight data and Priority Queue nodes respectively.
- Names: Algorithmic concepts like flight details and priority were coded as structured variable names, like flightnumber and priority.
- Altering Values: Memory management functions like malloc() and free() were used for dynamic allocation and deallocation, respectively. Examples include node creation in Add() and memory release in Finalize().
- Complicated Steps: Operations involving list traversal and conditional checks, like adding a flight in order based on departure time in Add(), were abstracted into their respective functions.
- And the answer is...!: Several functions return values, be it success status (as in Add()) or data (as in Count() functions).

Program Validation

Stress Testing Approach (See README.TXT)

- Ensured testing of edge cases such as adding duplicate flight numbers, handling invalid airline codes, and checking priority orders.

Validation Criteria

- Priority Queue must maintain flight orders based on flight departure time;
- All add, remove, and query operations should maintain the integrity of the queue;
- Special cases considered: invalid airlines, duplicate flight numbers, and priority conflicts.

Results

- All tests were successfully executed, verifying the accurate and robust operation of the priority queue. The data structure appropriately manages flights according to priority, even in edge cases.

Challenges and Improvements

Most Challenging Aspects

- Data Integrity: Ensuring that flights added or removed from the Priority Queue maintain their relative ordering based on departure time;
- Memory Management: Given that the Priority Queue relies heavily on dynamic memory allocation, managing memory without leaks and ensuring that freeing operations did not inadvertently modify the queue was challenging.

Key Improvements in Final Submission

- Enhanced Testing: Introduced additional test cases to validate the priority queue, demonstrating its resilience and correctness in various scenarios;
- Memory Management: Leveraged tools like valgrind to identify and fix potential memory leaks. This ensures optimal memory usage throughout the program's lifecycle.