National Teachers College
629 J Nepomuceno, Quiapo, Manila, 1001 Metro Manila
Bachelor of Science in Information Technology

Software Design & Analysis Document for **Data Structures**

# *UN SDG 12: Food Waste Management System*

**A Final Project**
Presented to the Faculty of the
College of Information Technology

In partial fulfillment
of the Course Requirements for the degree
of Bachelor of Science in Information Technology

**Submitted by BSIT-2.2:**

*Caneo, Dan Edryl*

*424003416*

*2025-2026*
*National Teachers College*

**Instructor:**
*Ms. Justin Louise Neypes*

**Date:**
*December 11, 2025*

# TABLE OF CONTENT

**I. Introduction**

1.1. Project Overview & UN SDG Target
1.2. Problem Statement

**II. Requirements & Analysis**

2.1. FRs and NFRs
2.2. Data Requirements
2.3. Complexity Analysis

**III. Design Specification**

3.1. Core Data Structures Used
- Justification
- Implementation

3.2. Algorithm Flowchart
3.3. Module Breakdown

**V. Conclusion**

5.1. Conclusion
5.2 Contributions

**VI. References**

# I. Introduction

### 1.1. Project Overview & UN SDG Target

This project is a C++ console application for managing a food inventory. Its main job is to help track food items and their expiry dates to reduce waste. When food is about to expire, the system makes an alert. Users can also log when food is thrown away to see patterns. This connects to UN SDG Goal 12, specifically Target 12.3, which wants to cut food waste in half by 2030 .

### 1.2. Problem Statement

In homes and businesses, food is often wasted because people forget what they have or don't see expiry dates coming. This system solves this by giving a clear list of all food, warning users about items expiring soon, and keeping a record of what was wasted and why. This helps users make better choices, buy less extra food, and waste less.

# II. Requirements & Analysis

**2.1. Functional Requirements and Non-Functional Requirements**

**Functional Requirements:**

- **FR1:**
  - The system must load data from files when it starts.
  - It must save data to files when it closes so this uses file handling

- **FR2:**
  - The system must let the user add, view, update and remove food items and this is the core function

- **FR3:**
  - The system must let the user log wasted food items and it must save these logs

- **FR4:**
  - The system must show alerts for food expiring soon within 7 days or less and this is a safety measure

- **FR5:**
  - The system must sort and display the waste logs and it must show which items are wasted the most (sort by quantity)

**Non-Functional Requirements:**

- **NFR1 (Performance):**
  - The system must work fast so sorting 50 waste records must take less than 1 second.

- **NFR2 (Robustness):**
  - The system must not crash from wrong user input and It must ask for the input again.

- **NFR3 (Maintainability):**
  - The code must be organized in modules and It must use classes and clear variable names.

### 2.2. Data Requirements

The system needs data to work. For testing, I created files with more than 50 records.

**Input Data:** The program reads these on startup:

- The file **inventory.dat** holds the food items.
- The file **alert.dat** holds the alert logs.
- The file **waste.dat** holds the waste logs.

**Data Structure:**

- **Each food item has:**
  - name (string)
  - quantity (int)
  - expiry data (string)
  - category (string)
  - days left until expiry (int)

- **Each alert log has:**
  - alert (string)
  - alerts (string)

- **Each waste log has:**
  - item name (string)
  - wasted quantity (int)
  - reason (string)
  - date wasted (string)

### 2.3. Complexity Analysis

The core algorithm is **Merge Sort** used for waste analytics.

- **Time Complexity: Merge Sort** has a time complexity of **O(n log n)**. This means the time it takes to sort grows slowly even if the number of waste logs (n) grows a lot. It is efficient for sorting.

- **Space Complexity: Merge Sort** needs extra space. Its space complexity is **O(n)**. This is because it creates temporary arrays while sorting. For this system with 1000 waste logs, this is acceptable.

- **Justification:**
    - I chose **Merge Sort** not merely as "an efficient algorithm" but as the optimal balance of **predictable performance**, **data integrity preservation**, and **educational value** for a system requiring reliable analytics on potentially large datasets.
    - **Merge Sort** guarantees **O(n log n)** comparisons for all cases through its divide and conquer methodology compared to **Heap Sort** with its unstable, complex heap operations, and **Quick Sort** with worst case **$O(n^2)$**, while acknowledging **Quick Sort's** performance advantages for in-memory sorting, **Merge Sort** provides **O(n log n)** performance that reliably satisfies the system's requirements.

# III. Design Specification

**3.1. Core Data Structures Used**:

I used three main data structures from our course, one from each term.

| **DSA Concept** | **Justification** | **Implementation** |
|---|---|---|
| Arrays (Prelim) | I used a static array to store the FoodItem objects. I chose an array because the inventory is a fixed size collection where I need fast access to any item by its index. It is simple and direct. I know the maximum number of items (1000) so a static array is okay. I used another array for WasteLog objects. | I declared them as global arrays: FoodItem inventory[MAX_ITEMS]; and WasteLog wasteLogs[MAX_WASTE];. I manage them with a counter variable like itemCount to know how many are actually filled. |
| Stacks (Midterm) | I used a Stack to manage alert messages. A Stack is Last In, First Out (LIFO). This is perfect for alerts because the most recent alert is the most important to show first to the user. | I created an AlertStack class. Inside, I used a private array of strings alerts[STACK_SIZE] and an integer top to track the last element. It has push() to add alerts and displayRecent() to show the last 5 alerts. When the stack is full, the oldest alert at the bottom is removed to make space (like a circular buffer). |
| Merge Sort (Finals) | I needed to sort the waste logs from highest quantity to lowest. This shows the user which items they waste the most. Merge Sort is an efficient, divide and conquer sorting algorithm. It is stable and has good O(n log n) performance. | I implemented it in the WasteAnalytics module. The functions merge() and mergeSortRecursive() work together. They break the wasteLogs array into smaller halves, sort them, and then merge them back together in sorted order. I sort by the quantity field in descending order. |

**3.2 Module Breakdown**

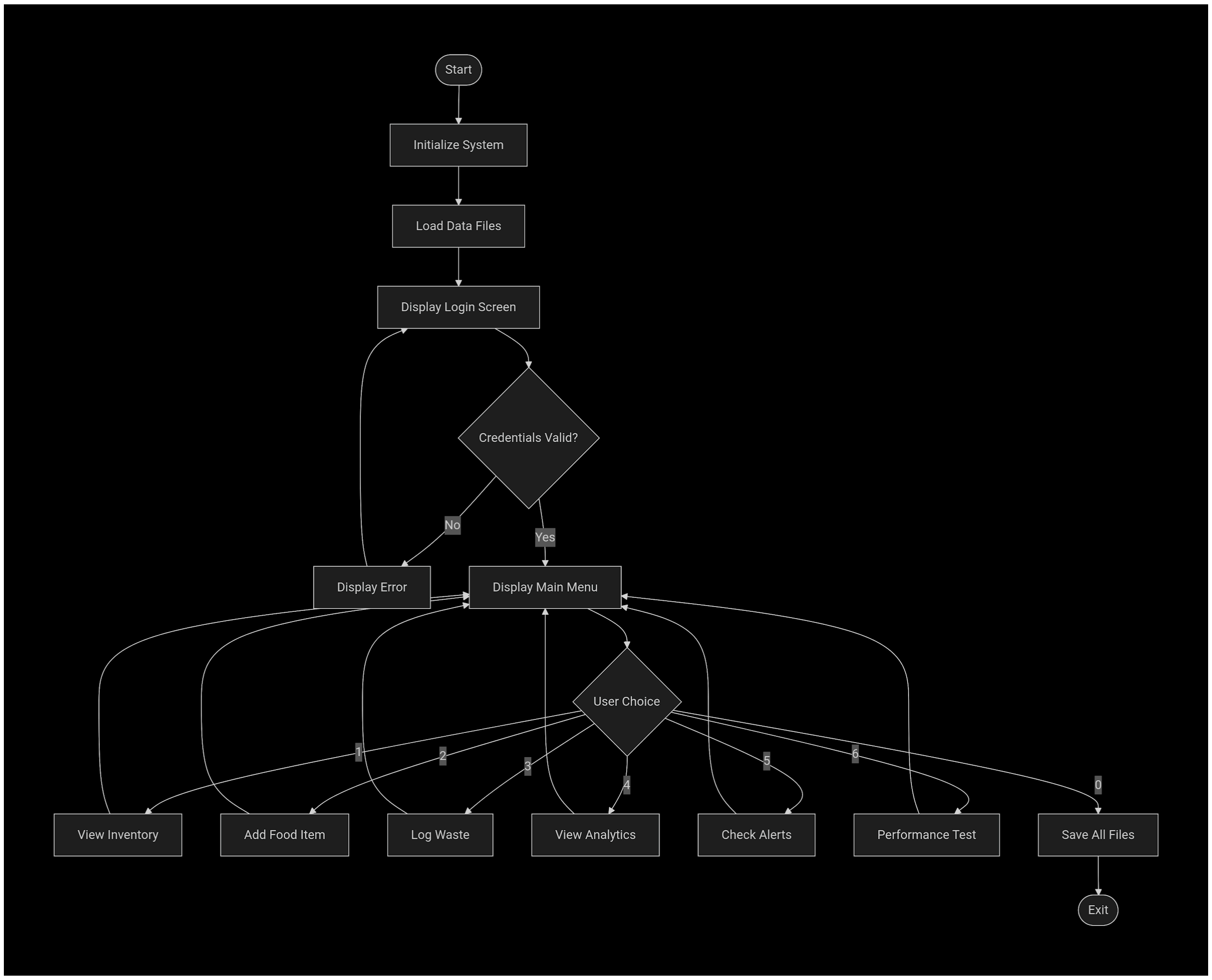
The system is divided into modules (files) for better organization.

- **Main (main.cpp):** Controls the program flow. Shows the menu and calls functions from other modules.

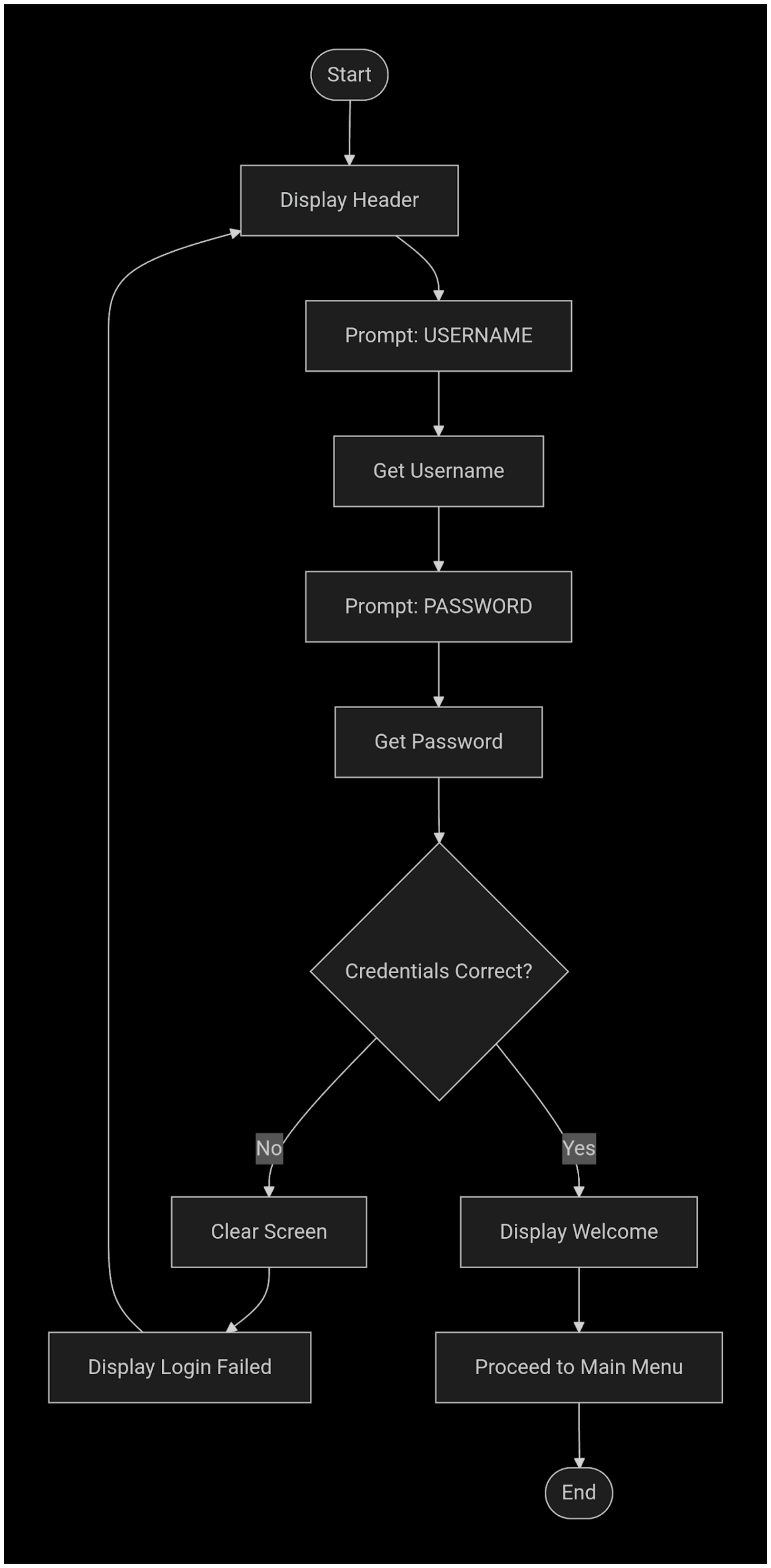
- **FoodArray Module (FoodArray.h/cpp):** Manages the inventory array. Contains all functions to add, remove, display, and search food items. Implements the ARRAY DSA.
- **AlertStack Module (AlertStack.h/cpp):** Manages the alert stack. Has push, pop, and display functions. Implements the STACK DSA.
- **MergeSort Module (MergeSort.h/cpp):** Manages waste logs. Contains the array of logs and the Merge Sort algorithm. Implements the MERGE SORT DSA.
- **Utilities Module (Utilities.h/cpp):** Contains common helper functions. Like getting the current date, calculating days to expiry, and input validation.

## 3.3 Algorithm Flowchart

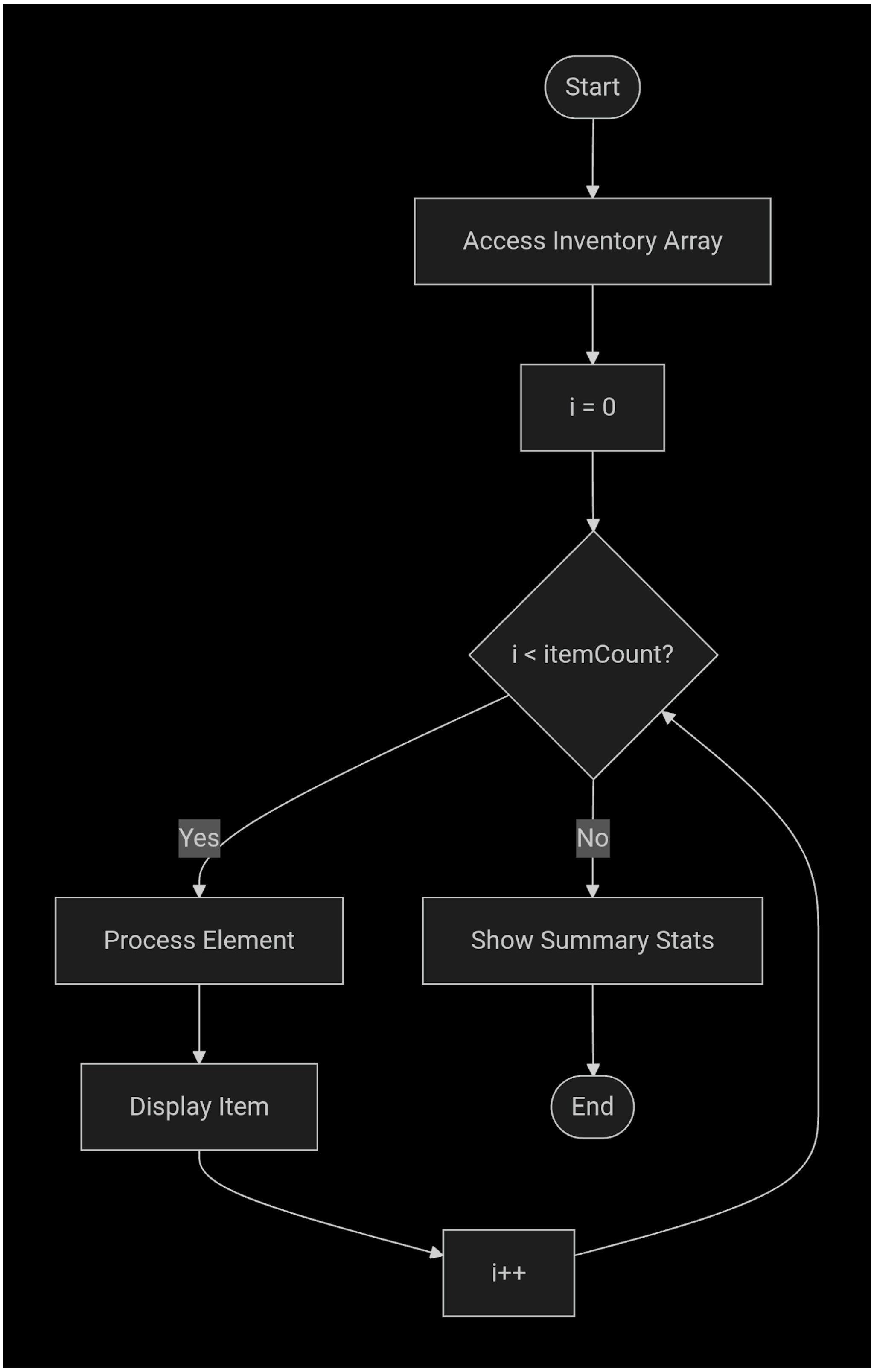
- **MAIN PROGRAM:**



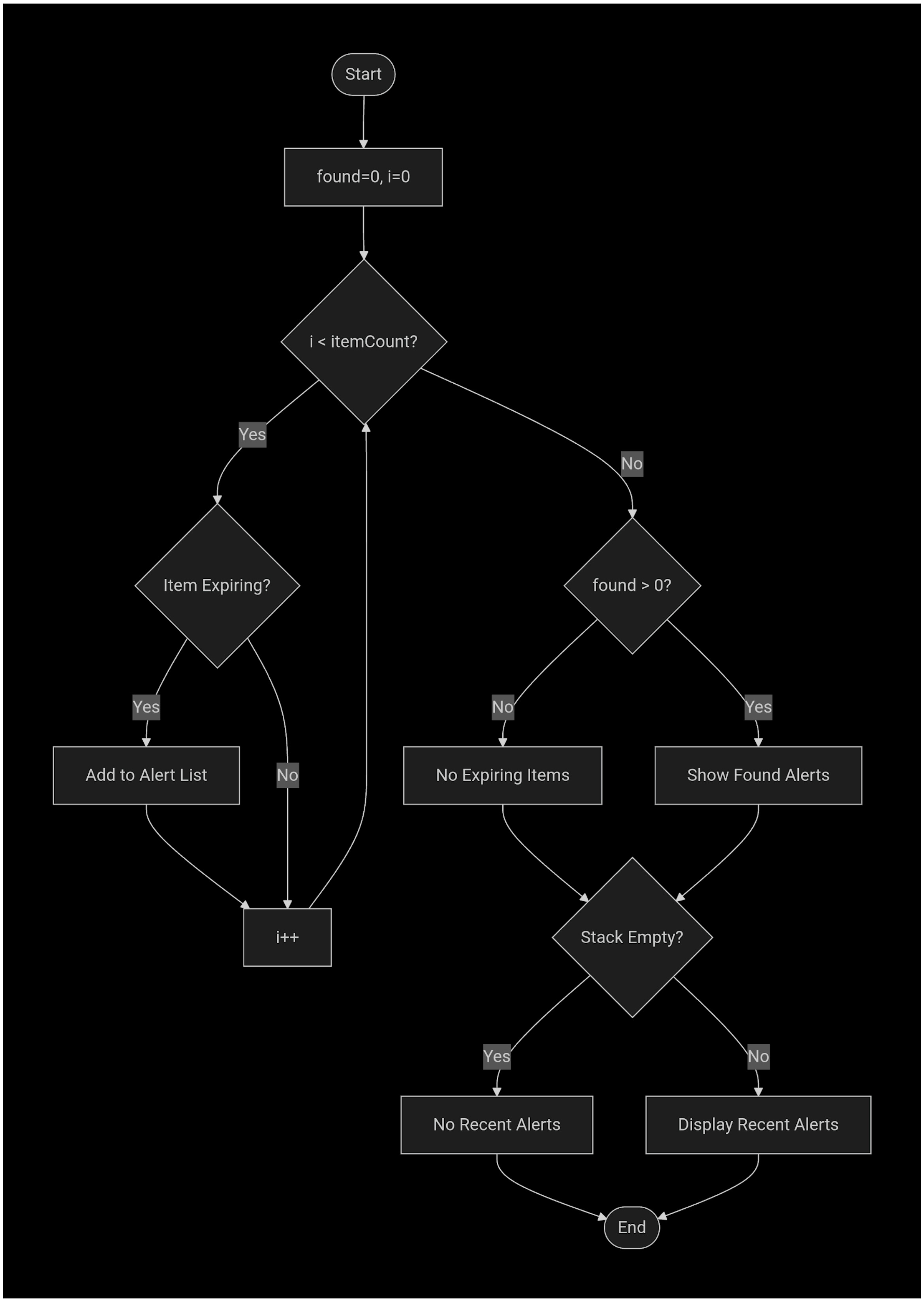
- **LOG IN PROCESS:**



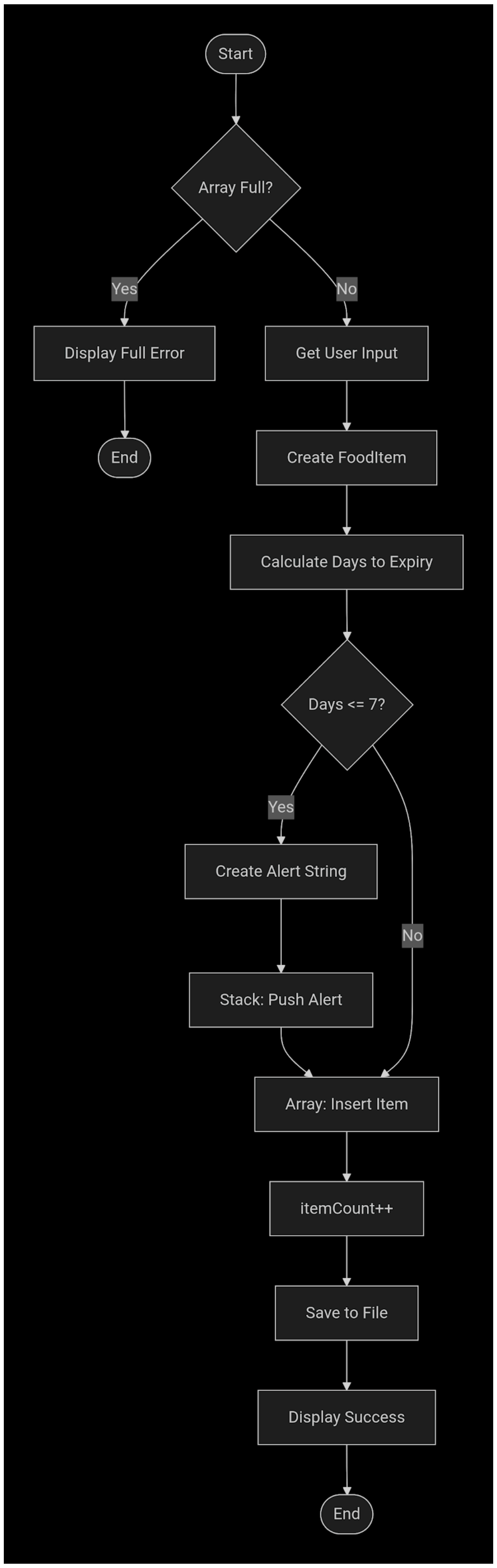
- **VIEW INVENTORY:**



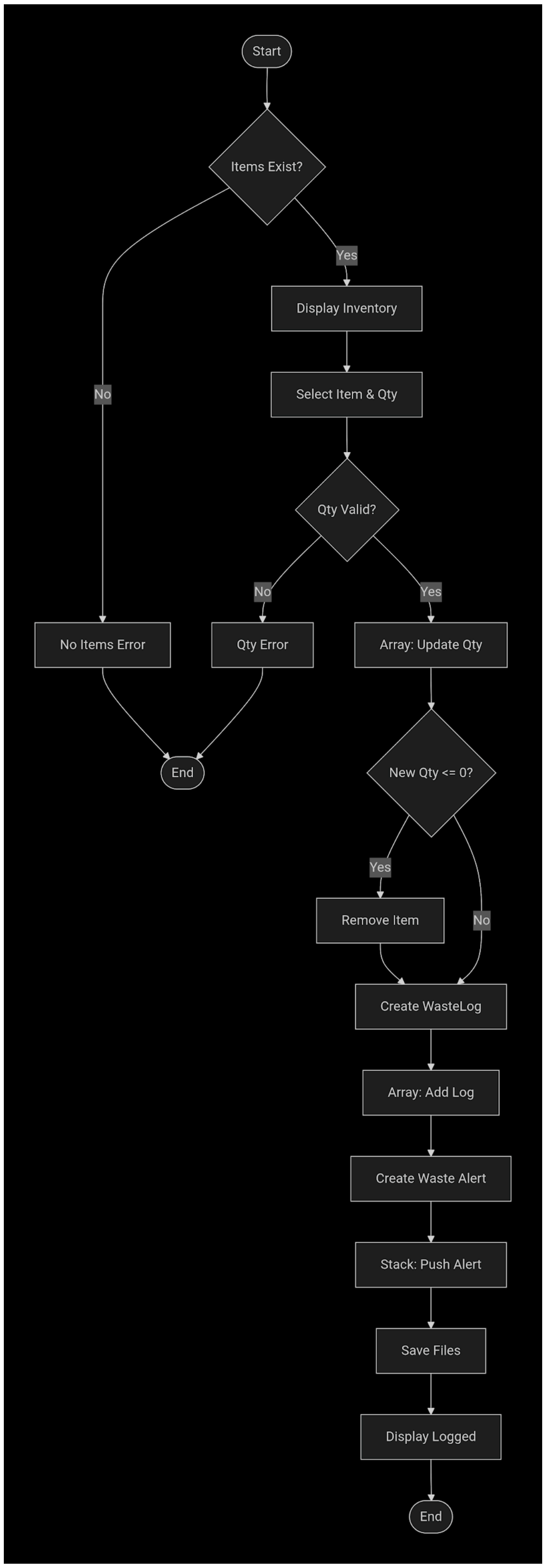
- **CHECK EXPIRY ALERTS:**



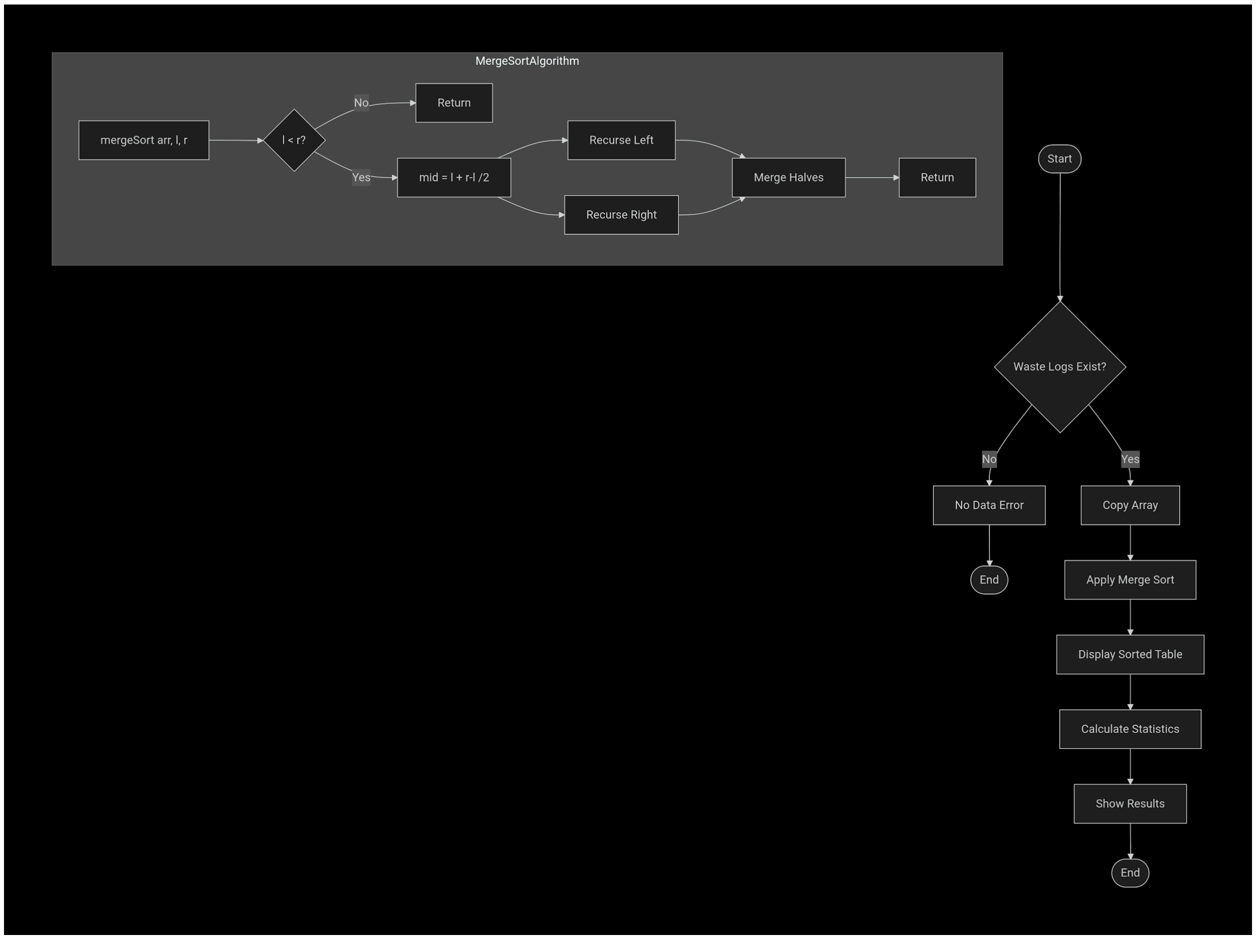
- **ADD FOOD ITEM:**



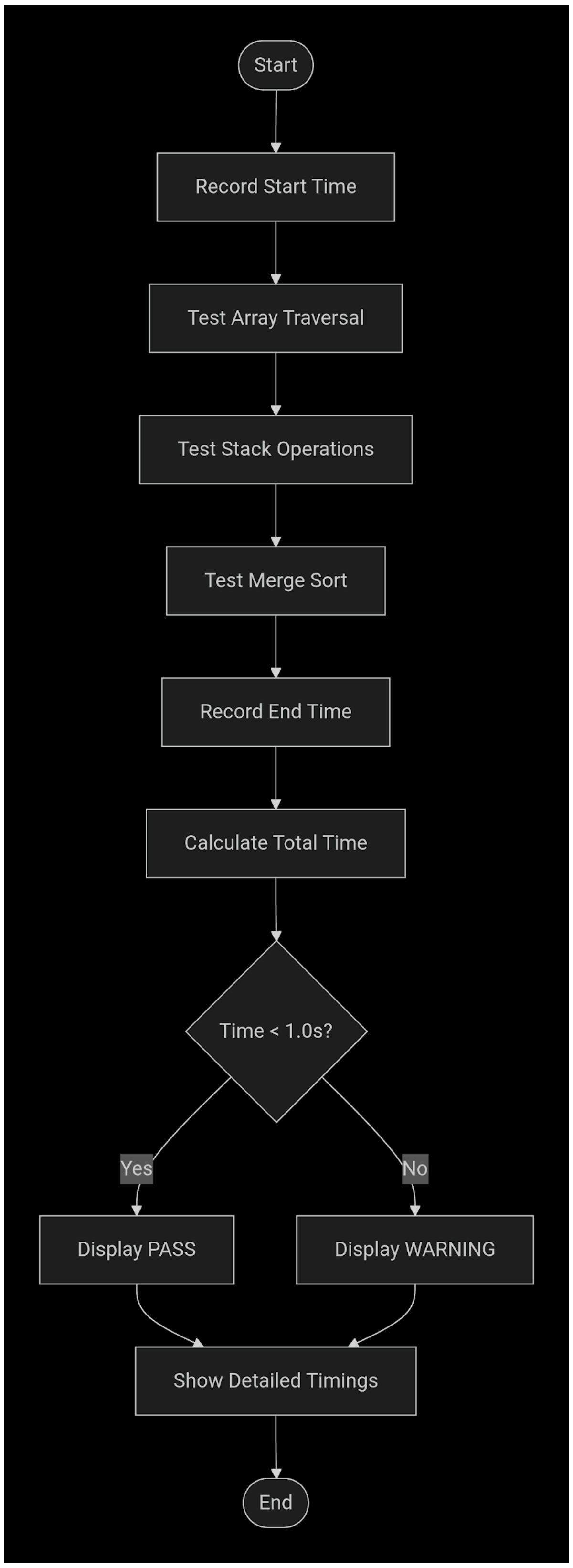
- **LOG WASTE FOOD:**



- **VIEW WASTE ANALYTICS:**



- **PERFORMANCE TEST:**



# V. Conclusion

### 5.1. Conclusion

I successfully built a Food Waste Management System. This project uses three main data structures I have learned: an **Array** for the food list, a **Stack** for alerts, and **Merge Sort** for the waste reports. The system lets you add food, see what's expiring, log waste, and view sorted analytics.

My thought process for the data structures went like this. For the inventory, I needed something simple where I could access items by a number. I thought about a Linked List but chose an **Array** because of it's random access. For alerts, I wanted the newest message to show up on top and the AI explained the **Stack** is good popping alerts.

The hardest part was the **Merge Sort** for the waste analytics. I needed to sort the waste logs from highest to lowest quantity. I knew simple sorting would be too slow for many records. The AI helped me understand the divide and conquer steps: split the list, sort each half, merge them back.

Debugging the **Merge Sort** was also tough because my program would freeze when sorting and the problem was an infinite recursion. I forgot the base case if (left < right) in the mergeSort function. Without it, the function would call itself forever and the AI helped me identify this by suggesting I trace the recursive calls with a small array.

In conclusion, It uses three key data structures: **Array**, **Stack**, and **Merge Sort**. The system helps track food inventory, reduce waste and it meets all the functional requirements. The code is organized into modules and it handles files and user input safely. This project helped me understand how to apply DSA concepts to solve a real world problem.

### 5.2 Individual Contributions

I made this project as a solo student.

- **Overall Design and Logic:** I partially defined the problem and planned the main features (inventory, waste logging, alerts, analytics).

- **C++ Implementation:** I wrote some of the C++ code in the modules (main.cpp, FoodArray.cpp, AlertStack.cpp, MergeSort.cpp, Utilities.cpp).

- **DSA Integration:** I implemented some of the Array operations, Stack class, and Merge Sort algorithm based on course lessons.

**Acknowledgement**: I used the DeepSeek AI assistant to help me with this project.
Specifically, I asked for help to:

- Organize the code into separate header and source files (modules).
- Choose the best variable names and data types for the FoodItem and WasteLog structs.
- Understand and implement the flow of data between the different modules.
- Get the correct logic for the Merge Sort algorithm on my specific data structure.
- Structure this SDAD document properly while the AI acted as a guide, but all final design decisions and some little bit of code implementations were done by me.

## VI. REFERENCES

https://www.geeksforgeeks.org/dsa/merge-sort/

https://www.programiz.com/cpp-programming/stack

https://www.programiz.com/cpp-programming/arrays

https://www.tutorialspoint.com/cplusplus/cpp_files_streams.htm

https://cplusplus.com/reference/

https://docs.mermaidchart.com/mermaid-oss/syntax/flowchart.html

https://mermaid.js.org/intro/syntax-reference.html