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

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.

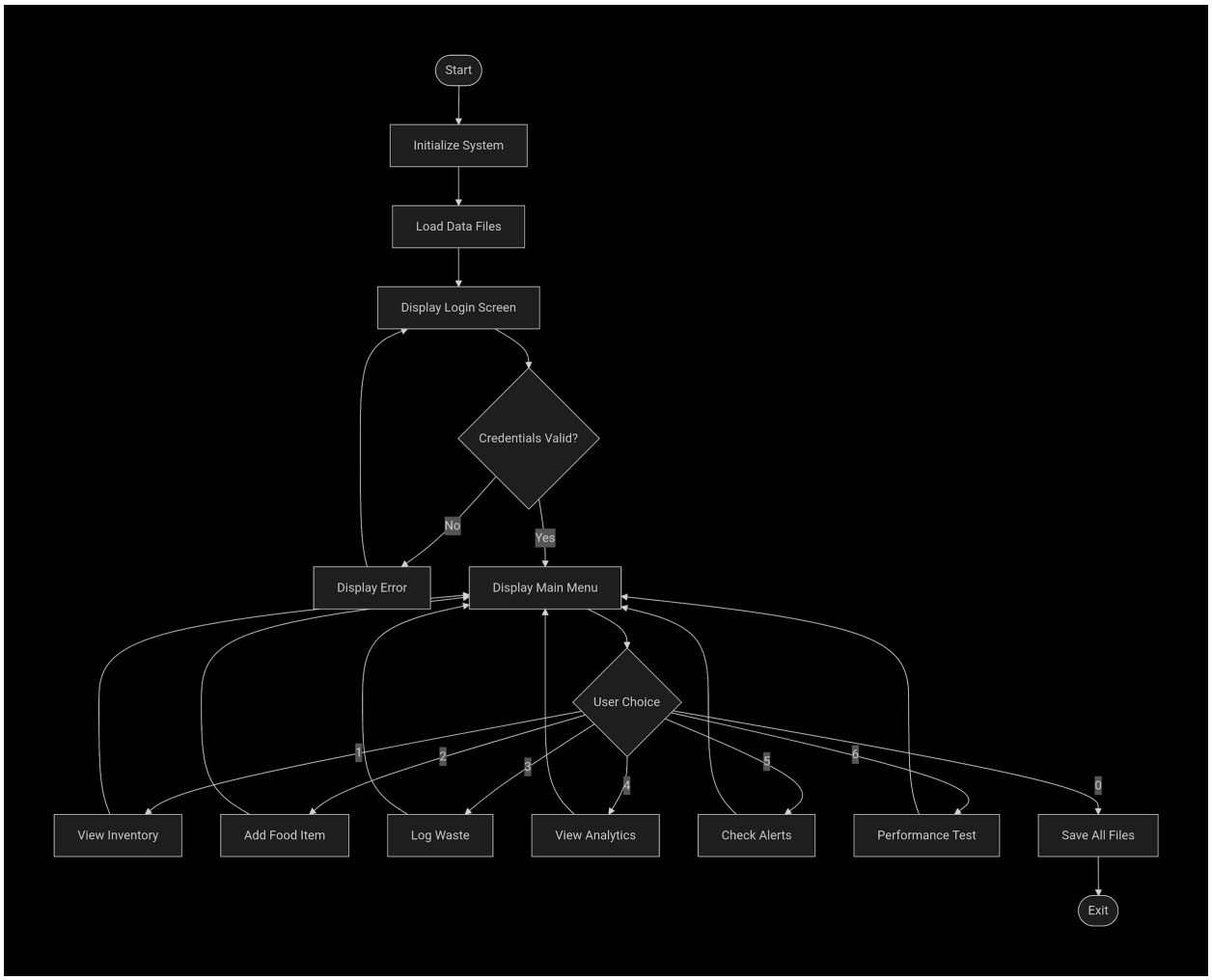
- **AlertStack Module (AlertStack.h/cpp):** Manages the alert stack. Has push, pop, and display functions. Implements the STACK DSA.

- **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:**




**EXPLANATION:**

### 1. START

- This is the beginning point.
- When the program's icon is double clicked, the computer finds this "Start" point and begins following the instructions.

### 2. INITIALIZE SYSTEM

This is the first set of instructions the program runs.

- The screen clears itself using the ANSI escape sequence.

- A welcome message shows up, saying the system is starting and mentioning the name of the system.
- The program looks inside itself and sets up empty "**slots**" in the computer's memory to hold the food inventory list and waste log list. These "**slots**" are the **ARRAYS** (**inventory[MAX_ITEMS]** and **wasteLogs[MAX_WASTE]**).
- Two important number trackers are set to zero: **itemCount** (how many foods are listed) and **wasteCount** (how many waste records exist).

**3. LOAD DATA FILES**

Now the program tries to remember the past sessions and it looks in the project folder for the **INPUT_DATA** folder.

- It tries to open **inventory.dat**. If the file exists, it reads the saved food list and puts it into the inventory **ARRAY**, and it also updates the **itemCount** number.
- It tries to open **waste.dat**. If the file exists, it reads the saved waste history and puts it into the waste logs **ARRAY**, and it also updates the **wasteCount** number.
- If the files don't exist, this step quietly does nothing and moves on and the arrays just stay empty.
- A message is printed to the screen saying how many items and logs were loaded.

**4. DISPLAY LOGIN SCREEN**

The program now shows the login page on the screen.

- It draws the banner with the system title and UN SDG info.
- It shows "**ADMINISTRATOR LOGIN**" and lines to separate the area.
- It displays "**USERNAME**: " and waits for an input.
- After the username input and press Enter, it displays "**PASSWORD**: " and waits again for an input.

**5. CREDENTIALS VALID?**

- This is a **DECISION** point (the diamond shape). The program compares the input typed.
    - It checks: Is the username exactly "**admin**" AND is the password exactly "**admin123**"?
- There are two possible paths from here:

- **NO PATH (Wrong Credentials):** If the username or password is wrong, the program goes to the "**Display Error**" step.

- **YES PATH (Correct Credentials):** If both are correct, the program moves forward to the "**Display Main Menu**" step.

**6. DISPLAY ERROR**

- This step is reached if the login fails.

    - The screen clears.
    - It shows a "**LOGIN FAILED**" message in red.
    - It tells the username/password was invalid and asks to try again.
    - It reminds the default login (**admin** / **admin123**).
    - After a short pause, the program's flow GOES BACK to step 4, "**Display Login Screen**". This creates a loop and this loop will happen forever, again and again, until they finally type the correct credentials.

**7. DISPLAY MAIN MENU**

- This step is reached after a successful login and this is the program's home screen.

    - The screen clears.
    - It shows the current date which is fetched from the phone or computer.
    - It shows the counts: "**INVENTORY: X ITEMS**" and "**WASTE LOGS: Y RECORDS**".
    - It calls the **displayRecent()** function from the **STACK** class. This function checks the top of the alert stack and shows the last 5 alert messages (like "**new item added**" or "**waste logged**").
    - It prints the list of options: **[1] View Inventory**, **[2] Add Food Item**, etc., all the way to **[0] Exit Program**.
    - Finally, it shows a prompt: "**ENTER YOUR CHOICE**: " and waits for an input.

**8. USER CHOICE**

- This is another major **DECISION** point.

    - It has seven possible paths (one for options 1-6, and one for option 0).

    - The path it takes depends entirely on the input and the choice directs the program's next action.

**9. THE MENU OPTIONS (Paths 1 through 6)**

- **PATH for Choice [1]: VIEW INVENTORY**
    - The program calls the **displayInventory()** function.
    - This function does a full **ARRAY** traversal. It uses a loop to go through every single item in the **inventory[]** array, from position 0 up to **itemCount-1**.
    - For each item, it calculates how many days are left until expiry (using the current date).
    - It formats all the data into a table with colored text (red for expired, yellow for soon, green for good).
    - It prints the table to the screen and then the flow goes back to the "**Display Main Menu**" step (Step 7) to show the menu again.
- **PATH for Choice [2]: ADD FOOD ITEM**
    - The program calls the **addFoodItem()** function.
    - It asks for details: **name**, **quantity**, **category**, **expiry date**.
    - This is an **ARRAY INSERTION** operation. It takes the new food data and places it into the next available empty spot in the **inventory[]** array, which is at index **itemCount**.
    - It then increases **itemCount** by 1.
    - It calculates the days to expiry and if the food expires in 7 days or less, it creates an alert string.
    - **STACK OPERATION (PUSH**): It takes that alert string and pushes it onto the **AlertStack** by calling **alertStack.push(alert)**.
    - It calls **saveInventory()** to write the entire updated **ARRAY** to the **inventory.dat** file.
    - Shows a success message and the flow returns to "**Display Main Menu**".
- **PATH for Choice [3]: LOG WASTE**

- The program calls the **logWaste()** function.
- First, it shows the inventory so you can pick an item.
- You choose an item number and how much was wasted.
- **ARRAY UPDATE**: It finds that item in the **inventory[]** array and subtracts the wasted quantity from its quantity value.
- **ARRAY INSERTION**: It creates a new WasteLog record and adds it to the wasteLogs[] array at index wasteCount, then increases wasteCount.
- **STACK OPERATION (PUSH**): It creates a waste alert and pushes it to the **AlertStack**.
- If the item's new quantity is zero, it performs an **ARRAY DELETION WITH SHIFT**. It removes that item from the middle of the array by shifting every item after it one position to the left.
- It calls **saveInventory()** and **saveWasteLogs()** to update both data files.
- Flow returns to "**Display Main Menu**".

- **PATH for Choice [3]: LOG WASTE**
  - The program calls the **logWaste()** function.
  - First, it shows the inventory so you can pick an item.
  - You choose an item number and how much was wasted.
  - **ARRAY UPDATE**: It finds that item in the **inventory[]** array and subtracts the wasted quantity from its quantity value.
  - **ARRAY INSERTION**: It creates a new **WasteLog** record and adds it to the **wasteLogs[]** array at index **wasteCount**, then increases **wasteCount**.
  - **STACK OPERATION (PUSH**): It creates a waste alert and pushes it to the **AlertStack**.
  - If the item's new quantity is zero, it performs an **ARRAY DELETION WITH SHIFT**. It removes that item from the middle of the array by shifting every item after it one position to the left.

- It calls **saveInventory**() and **saveWasteLogs()** to update both data files.

- Flow returns to "**Display Main Menu**".

- **PATH for Choice [4]: VIEW ANALYTICS**

  - The program calls the **viewWasteAnalytics()** function.

  - This function first creates a temporary copy of the **wasteLogs[]** ARRAY.

  - **MERGE SORT EXECUTION**: It calls **mergeSort(tempArray, 0, wasteCount-1)**. This is the most complex part.

  - **Merge Sort** is recursive. It keeps splitting the temporary array into smaller halves until each half has only one element.

  - Then it starts merging them back together in sorted order (by quantity, from highest to lowest). The **merge()** function does this by comparing elements and building the sorted list.

  - After sorting, it simply does a traversal of this now-sorted temporary array and prints it as a table.

  - It also prints how long the sorting took and the flow returns to "**Display Main Menu**".

- **PATH for Choice [5]: CHECK ALERTS**

  - This path is a bit different as it doesn't just show the stack alerts from the main menu and instead, it calls **checkExpiryAlerts()**.

  - This function does an **ARRAY SEARCH AND FILTER**. It traverses the **inventory[]** array and picks only the items where **daysToExpiry <= 7**.

  - It prints a special table containing just those urgent items and the flow returns to "**Display Main Menu**".

- **PATH for Choice [6]: PERFORMANCE TEST**

  - The program calls **runPerformanceTest()**.

  - It uses a timer to measure how long it takes to run **displayInventory()** (array traversal of all items).

- It uses a timer to measure how long it takes to run **mergeSort()** on all waste logs.
- It adds up the times and checks if the total is less than 1 second.
- It prints the results and the flow returns to "**Display Main Menu**".

- **THE EXIT PATH for Choice [0]: SAVE ALL FILES**
  - This is the exit sequence and the program does not go back to the main menu.
  - It clears the screen and shows a "**saving data**" message.
  - It calls **saveInventory()** one last time to write the current **inventory[]** array to the **.dat** file.
  - It calls **saveWasteLogs()** one last time to write the current **wasteLogs[]** array to the **.dat** file.
  - The flow then moves to the final step.

**10. END**

- This is the termination point and after saving the files, the program prints a goodbye message.
- Then, it shuts itself down completely, the console window closes, and the process that was started at the "BEGIN" point is now finished.

# 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