I. Introduction

1.1. Project Overview & UN SDG Target

The Blood Donation Management System (BDMS) is a high-efficiency inventory management system designed to optimize the allocation and use of perishable blood units. The core challenge in blood management is minimizing wastage due to expiration, which requires precise tracking and dispatching based on priority.

This project directly supports the United Nations Sustainable Development Goal (SDG) 3: Good Health and Well-being, specifically addressing Target 3.8: Achieve universal health coverage... by ensuring essential health services and access to safe, effective, quality, and affordable essential medicines and vaccines for all.

Our system fulfills this objective by focusing on the following critical functional requirement:

- Functional Requirement 3 (FR3): The system must efficiently prioritize and identify the blood unit closest to its expiration date for dispatch.

This prioritization is achieved through the use of an advanced data structure—the Min Priority Queue (Min Heap)—guaranteeing that the blood unit with the most urgent need for dispatch is identified in highly efficient O(log n) time complexity, significantly reducing inventory expiration and improving public health resource management.

1.2. Problem Statement (What real-world problem does the app solve?)

The Blood Donation Management System (BDMS) is designed to solve a critical and costly problem in public health: the expiration of donated blood units before they can be used.

The Problem

- Perishability and Waste: Red blood cells typically have a limited shelf life of approximately 42 days from collection. Poor inventory management often leads to tragic waste, where units expire while sitting in storage due to non-optimal dispatching decisions. This wastage undermines public trust, squanders volunteer effort, and imposes significant financial and logistical burdens on healthcare systems.

- The Dispatching Challenge: Traditional inventory methods (like First-In, First-Out, or FIFO) do not account for the varying expiration dates and are often slow to manage. The challenge is complex because the system must constantly track thousands of unique units and instantaneously determine which single unit carries the highest priority for dispatch at any given moment.

The Solution (DSA Justification)

The BDMS addresses this by implementing a Min Priority Queue (Min Heap) where the priority key is the unit's Expiration Date.

This structure ensures that the system satisfies Functional Requirement 3 (FR3):

- The Min Heap enforces a system where the unit with the earliest (smallest) expiration date is automatically placed at the root of the heap.
- The most urgent unit can be identified and extracted in O(log n) time, providing the rapid decision support necessary to dispatch the closest-to-expiring blood unit before it becomes unusable.

II. Requirements & Analysis

2.1. Functional Requirements and Non-Functional Requirements (List of features, e.g., FR1, FR2)

Functional Requirements (FRs)

Functional Requirements define the *specific features* and *behaviors* the system must perform to meet the objectives outlined in the problem statement.

ID	Description	DSA Used	Fulfillment
FR1	Data Loading and Parsing: The system must read unit data from a CSV file and load it into an internal memory structure.	Linked List (DataParser)	Data is parsed line-by-line and stored in dynamic nodes, allowing for efficient, continuous insertion of records.
FR2	Data Encapsulation: The system must store each unit's information (ID, Expiration Date, Blood Type) as a single object.	BloodUnit Class	Achieved using a C++ class with public attributes and a custom constructor to ensure data integrity.
FR3 (Core)	Priority Dispatch: The system must instantly identify and dispatch the blood unit that is closest to its expiration date to minimize waste (SDG 3).	Min Priority Queue (Min Heap)	Achieved via the extractMin() operation, which finds and removes the highest priority item in O(log n) time.
FR4	User Interface: The system must provide	main.cpp	Implemented using a simple console loop

	a console-based menu allowing users to select actions (Dispatch, Search, Exit).		and switch statements for command navigation.
FR5	Inventory Search: The system must allow users to search for a specific BloodUnit using its unique ID.	Binary Search	Achieved by sorting the inventory and using a Binary Search algorithm to locate the unit in O(log n) time.

Non-Functional Requirements (NFRs)

Non-Functional Requirements define the *quality attributes* of the system (how well it performs, how easy it is to maintain, etc.).

ID	Description	Fulfillment Strategy
NFR1	Performance/Efficiency: Core operations (Dispatch and Search) must operate with a highly efficient time complexity.	All major functional requirements are implemented using O(log n) Data Structures (Min Heap and Binary Search) to ensure near-instantaneous response times, regardless of the inventory size.
NFR2	Robustness: The system must be capable of handling unexpected inputs (e.g., non-numeric volume data) and exceptions (e.g., attempting to dispatch from an empty inventory).	Implemented try-catch blocks in DataParser (for file parsing) and extractMin() (for empty queue checks) to prevent system crashes.
NFR2	Maintainability/Modularity: The code structure must be modular, separating data handling, core algorithms, and the main application logic.	The system is divided into clear C++ classes (DataParser, MinPriorityQueue, InventorySearch) with distinct responsibilities,

		adhering to object-oriented programming (OOP) principles.

2.2. Data Requirements (Description of input data structure and size)
The Blood Donation Management System (BDMS) relies on external input to populate its inventory. This section details the structure and constraints of the data processed by the system.

A. Input Data Structure (data.csv)

The system accepts input from a Comma Separated Values (CSV) file, typically named data.csv. Each line in the file represents a single blood unit record, organized by six critical fields:

Field	Data Type	Description	Constraints
UnitID	String	Unique identifier for the blood unit (e.g., BU001).	Must be unique and non-empty.
Blood Type	String	The ABO/Rh blood group (e.g., O+, AB-).	Must match a valid blood type format.
Volume	Integer	The volume of the unit in milliliters (mL).	Must be a positive integer value (NFR2: Robustness handles parsing).
Collection Date	String	Date the blood was drawn (YYYY-MM-DD).	Used for auditing purposes.
Expiration Date	String	The Priority Key (YYYY-MM-DD).	Crucial: Used by the Min Heap to determine dispatch priority (earlier date = higher priority).
CenterID	String	ID of the donation center.	Used for tracking location.

B. Internal Data Structure (BloodUnit Class)

To enforce FR2: Data Encapsulation and NFR3: Modularity, each record from the CSV file is translated into an instance of the C++ BloodUnit class.

- The BloodUnit class acts as a single, self-contained entity that holds all six attributes.

- This object is the node payload for both the Linked List (during parsing) and the Min Heap (during dispatching).

C. Data Size Constraints

The system is designed to handle large datasets efficiently.

- Average Size: Typical input files contain between 100 to 1,000 records (blood units).
- Performance Constraint: Due to the implementation of the core algorithms (Min Heap and Binary Search) in O(log n) time complexity, the BDMS can efficiently manage an inventory scaling up to 50,000+ records without experiencing noticeable degradation in Dispatch or Search performance.

2.3. Complexity Analysis: Expected Time/Space complexity of the Core Algorithm (justify using Big O notation).

The BDMS satisfies the NFR1 (Performance/Efficiency) requirement by utilizing advanced Data Structures and Algorithms with optimal Big O time complexity. The following analysis focuses on the time required to perform the most critical operations, where n is the number of blood units in the inventory.

A. Min Priority Queue (Min Heap) Analysis

The Min Heap is the core algorithm fulfilling FR3 (Priority Dispatch). It prioritizes units by the shortest time until expiration.

Operation	Time Complexity (Big O)	Justification
Insertion (insert)	O(log n)	When a unit is added, the heapifyUp operation ensures the heap property is restored. This operation involves at most swapping the new element from the bottom up to the root, moving only along the height of the binary tree. The height of a balanced binary tree is log_2 n.
	O(\log n)	This operation removes the root (highest priority) and restores the heap property using heapifyDown. This process also moves along the height of the tree, requiring a logarithmic number of

		comparisons and swaps to re-establish the correct order.

III. Design Specification

3.1. Core Data Structures Used (The Five): For each of the five required DSA concepts, include a section detailing: Justification: Why was this specific DSA chosen for its role? Implementation Details: How did you implement it (e.g., adjacency list for Graph, array for Heap)?

This project's architecture is built upon the strategic integration of five core Data Structures and Algorithms (DSAs) to maximize efficiency, robustness, and solve the critical problem of priority dispatching.

1. BloodUnit Class (Data Encapsulation)

- Justification: Chosen to satisfy FR2 (Data Encapsulation). Encapsulating all six unit attributes (ID, dates, volume, etc.) into a single, clean object simplifies data handling, ensures data integrity, and makes the system modular for passing units between different classes (e.g., from the DataParser to the MinPriorityQueue).
- Implementation Details: Implemented as a standard C++ class (BloodUnit) with public member variables and a parameterized constructor to initialize an object immediately upon reading a line from the CSV file.

2. Linked List (Prelim DSA: Data Parsing)

- Justification: Chosen for its simplicity and efficiency in the initial data loading phase (FR1). Since data is read line-by-line, the Linked List's ability to perform O(1) insertion at the head of the list makes the parsing process extremely fast and dynamically memory-efficient.
- Implementation Details: Implemented using a Singly Linked List structure. A Node struct was defined, containing a BloodUnit object (data) and a pointer (next) to the next node. The DataParser class maintains a single head pointer.

3. Vector (Array-based Implementation)

- Justification: Chosen as the underlying structure for implementing both the Min Heap and for preparing the data for the Binary Search. The std::vector offers contiguous memory allocation, which is critical for efficient index-based access in Heap operations and for the required sorting/searching functions.
- Implementation Details: Used as a C++ std::vector<BloodUnit> member variable within both the MinPriorityQueue class (to store the heap structure) and the InventorySearch class (to hold the sorted data for Binary Search).

4. Min Priority Queue (Min Heap)

- Justification: This is the core algorithmic choice to satisfy FR3 (Priority Dispatch). The Min Heap guarantees that the single item with the highest priority (the blood unit closest to expiration) can be extracted in O(log n) time. No other structure guarantees this retrieval speed while maintaining order for the entire inventory.
- Implementation Details: Implemented using a Vector (std::vector<BloodUnit>). Priority is determined by the custom comparator function, isEarlier(), which uses string comparison on the ExpirationDate field. The heap integrity is maintained via two helper functions: heapifyUp() and heapifyDown().

5. Binary Search Algorithm

- Justification: Chosen to satisfy FR5 (Inventory Search) due to its exceptional O(log n) time complexity. This performance is essential for quickly locating any specific blood unit in a potentially massive inventory, aligning with NFR1 (Performance).
- Implementation Details: The search is performed on the std::vector<BloodUnit> that has been sorted by the UnitID. The algorithm iteratively checks the middle element of the current search range, reducing the search space by half in each step until the target ID is found or the range is exhausted.

IV. Contribution

Name	Primary Role / Module
Roldan, Christian Maverick Z.	Lead Integrator, MinPriorityQueue (Min Heap) Logic, Final Testing/Debugging
Musngi, Trishia	DataParser (Linked List) Implementation, SDAD Design/Formatting
Comcom, John Vincent	InventorySearch (Binary Search) Implementation, Flowchart Design/Diagram
Nivales, Ande Jexer	BloodUnit Class Implementation, README/Usage Documentation
Soriano, Mark	Input/Output Handling & Menu Logic (main.cpp), Final Data Validation

V. Conclusion

The Blood Donation Management System efficiently manages blood inventory using optimized data structures, reducing wastage and improving dispatch accuracy. Through the integration of Min Heap, Linked List, Binary Search, and modular class design, the system achieves both high performance and strong alignment with SDG 3 objectives.