KING MONGKUT'S UNIVERSITY OF TECHNOLOGY THONBURI

FINAL PROJECT REPORT CPE 112
DEPARTMENT OF COMPUTER ENGINEERING
FACULTY OF ENGINEERING

Project Title	To-Do List Management System	
Group Title	Amaykelpar	
Member(s)	Mr. Sai Bhone Myint Myat	67070503479
	Mr. Moe Htet Aung	67070503482
	Ms. Thu Thu Wai	67070503486
Program	Bachelor of Engineering	
Field of Study	Computer Engineering	
Department	Computer Engineering	
Faculty	Engineering	
Academic Year	2024-2025	

10 pages below

1. INTRODUCTION

1.1 Problem Statement

Task management is a common challenge faced by students, professionals, and all of us who deal with multiple tasks. Paper based to-do list methods often fail when it comes to organizing tasks by categories, prioritizing deadlines, recovering mistakenly deleted tasks, or statistical view. There is a need for an efficient, structured system that supports categorized task handling, automatic prioritization, deadline awareness, and reliable data persistence.

1.2 Objective

The objective of this project is to design and implement a **Task Management System** in the C programming language that:

1. Allows create, update, delete, and restore categorized tasks.
2. Implements sorting and viewing of tasks by deadline, priority, and alphabetical order.
3. Offers detailed task statistics and views based on task status and time-sensitivity.
4. Ensures data persistence through CSV file.

1.3 Scope

1.3.1 Functional Scope

1. Task Categorization: Supports grouping of tasks under categories.
2. Priority Management: Utilizes *priority queue* to manage and sort tasks.
3. Task Restoration: Allows soft-deletion and *stack-based* undo functionality.
4. Sorted Viewing: Provides sorting options by name, deadline and priority.
5. Detailed Views: Provides viewing of full task descriptions and category.
6. Statistical Overview: Displays completion, pending or in-progress statistics.
7. Data Persistence: Reads from and updates to *tasks.csv* file.

1.4 Organization

1. task_manager.c/h: Core task management functionality
2. priority_queue.c/h: Implementation of the priority queue for task sorting
3. file_handler.c/h: CSV file operations for data persistence
4. ui.c/h: User interface components and display functions
5. utils.c/h: Utility functions for input validation and other helpers
6. main.c: Main program flow and menu system

2. ARCHITECTURE OVERVIEW

2.1 Data Structures Uses

Data Structure	Purpose	Usage
Priority Queue	To maintain tasks in order of priority and due date	Array-based binary heap
Stack	To manage and restore deleted tasks (LIFO)	Used for “Restore Tasks”
Circular Linked List	To enable alphabetical task sorting	Applied in "SortByName"
Singly Linked List	To manage task categories and their respective task lists	Structure to organize tasks
Array		All through the system

2.2 Why These Data Structures Were Used

2.2.1 Priority Queue (Array-based Binary Heap)

We choose this for its efficiency in automatically maintaining task order by priority and due date. A heap can operate fast insertions [*O(log n)*] and access to the highest-priority task [*O(1)*] that make it different to linear structures like arrays or linked-list.

2.2.2 Stack (LIFO)

We used for "Restore Task" because deleted tasks should be restored in reverse order of deletion. Stack's *LIFO* behavior makes it the ideal structure which is more simpler and more efficient than others.

2.2.3 Circular Linked List

We used a circular linked list for alphabetical sorting because it traversal from the end back to the start can able to continue iteration without special boundary checks. This simplifies the sorting and display process and making it more efficient and compared to arrays or standard linked lists.

2.2.4 Singly Linked List

We used for managing categories because it provides efficient insertion and deletion. Its simplicity make ideal for list of categories.

2.2.5 Array

Arrays are used throughout the system for tasks like displaying data, sorting, collecting input, and pairing categories with tasks. Their fast access and simple structure make for temporary storage and efficient iteration during task viewing and UI rendering.

3. CODE WALKTHROUGH AND FEATURES

3.1 Main Program (main.c)

- Role: Central control flow and menu navigation.
- Relevant Data Structures: Singly Linked List

3.2 Task Manager (task_manager.c/ .h)

- Role: Core logic is to create, update, delete, restore, and view tasks.
- Data Structures: *Priority Queue* – used in each Category to store tasks by priority and deadline.
  Stack – soft-deleted tasks are stored in deletedStack for restoration.
  Circular Linked List – implement in sortTasksByName() using a circular structure to sort tasks alphabetically.

3.3 Priority Queue (priority_queue.c/ .h)

- Role: Maintains task order within a category by priority and due time.
- Data Structure: Binary Heap (Array-based) for efficient insertions.

3.4 File Handler (file_handler.c/ .h)

- Role: Data persistence (load and save tasks).
- Data Structures: Linked List of Categories and their internal Priority Queues.

3.5 User Interface (ui.c/ .h)

- Role: Styled terminal output.
- Data Structures: Arrays of tasks and parallel category arrays for display.

3.6 Utility Functions (utils.c/ .h)

- Role: Handles input, validation, and formatting.
- Data Structures: No major DS, but crucial for supporting time-sensitive validation.

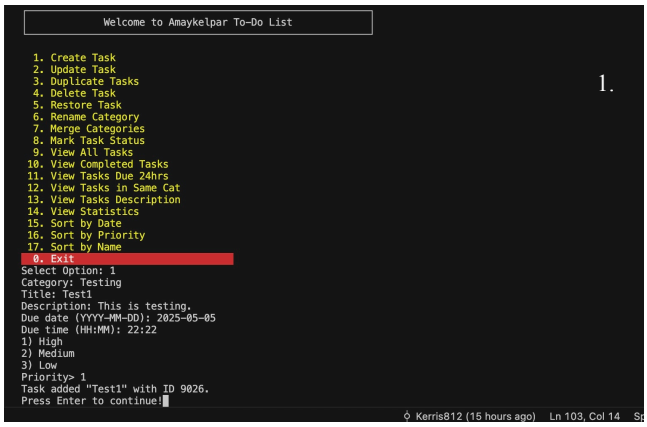
3.7 Header

We used *Header* to organize our code by separating declarations from implementations. Using *Header* promote our code to debug easier. In this project, *Header* allow us different modules to interact seamlessly and make ensure consistency, enabling clean and structured code across multiple source files.

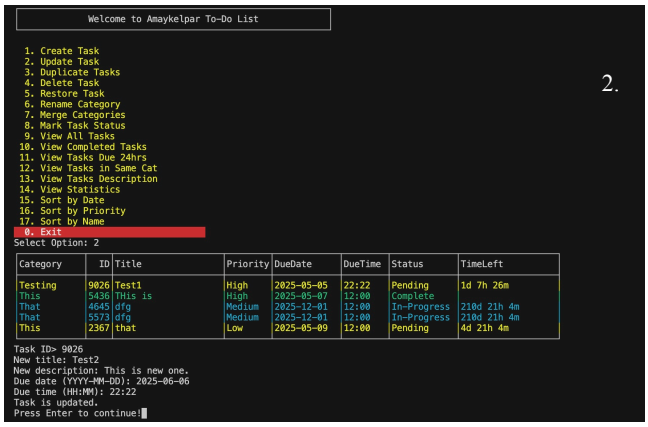
3.8 Detailed Features

3.8.1 Task Management Features

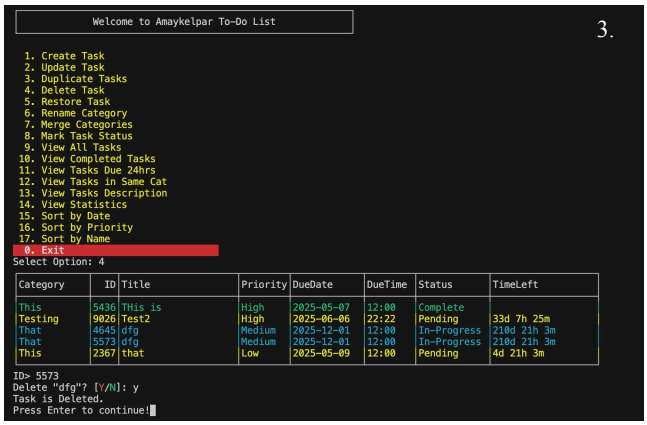
1. Task Creation: Users can create tasks with title, description, due date (no valid date or past date is not allowed), due time (no valid or time within 30 mins from now is not allowed), and priority level.
2. Task Updates: Existing tasks can be updated with new information.
3. Task Deletion: Tasks can be removed from the list with two-step verification.
4. Task Restoration: Deleted tasks can be restored if needed.
5. Task Duplication: Existing tasks can be duplicated to create similar tasks quickly.



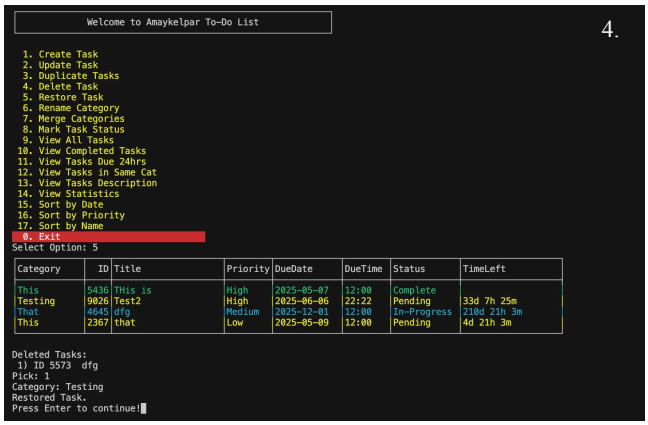
1.



2.



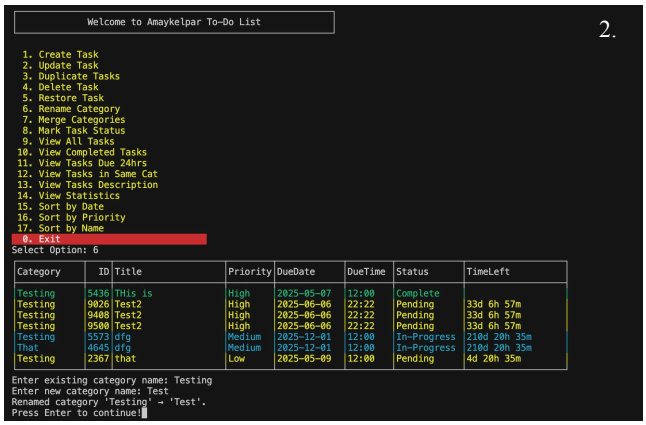
3.

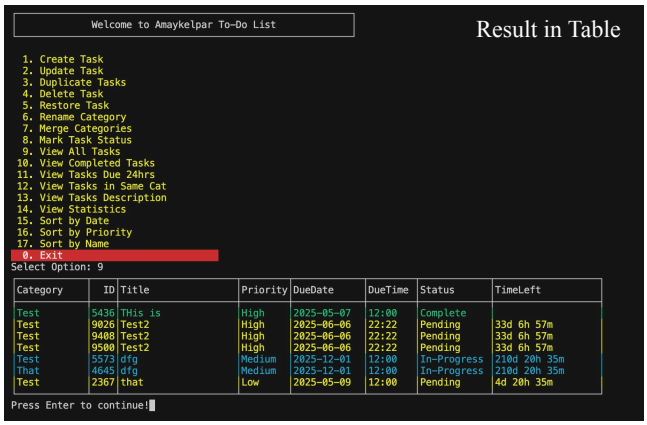


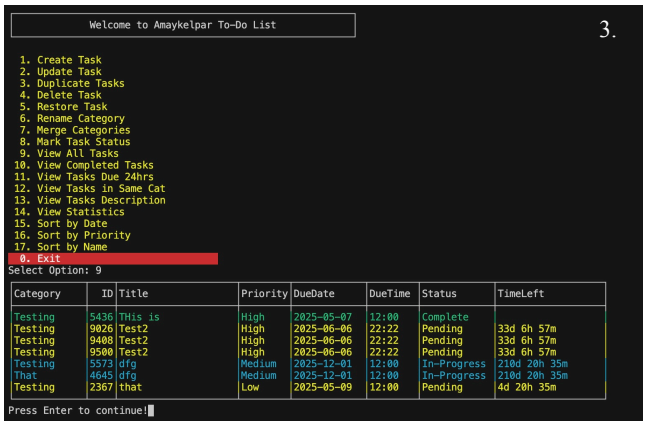
4.

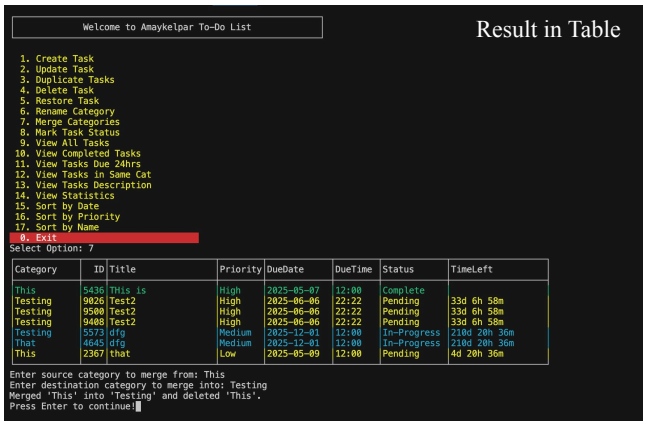
3.8.2 Organization Features

1. Category Management: Tasks are organized into each categories.
2. Category Renaming: Categories can be renamed. If the new name entered already exists in the system, the operation is aborted; else, the category name is successfully updated.
3. Category Merging: Two categories can be merged into one.



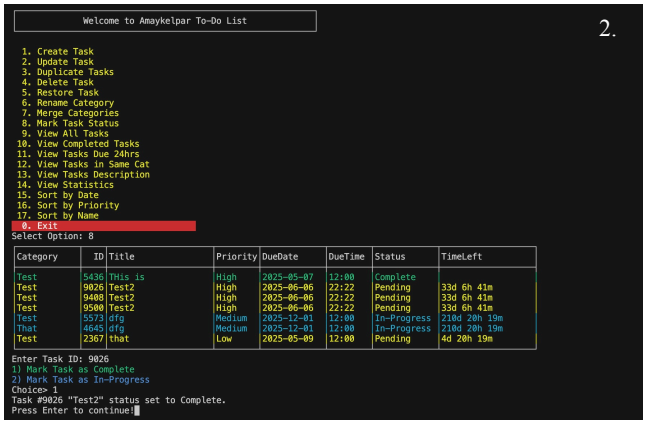






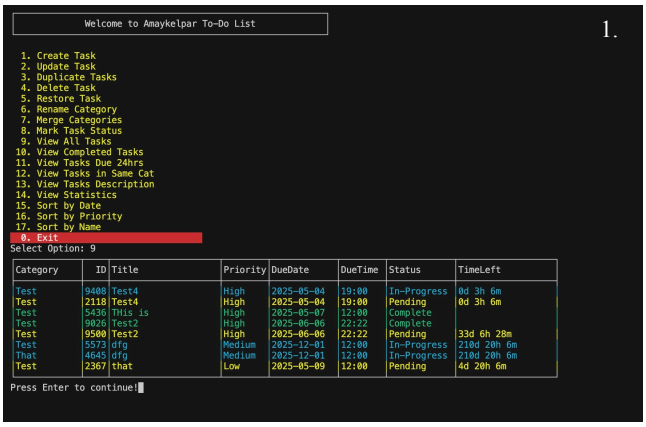
3.8.3 Status Management

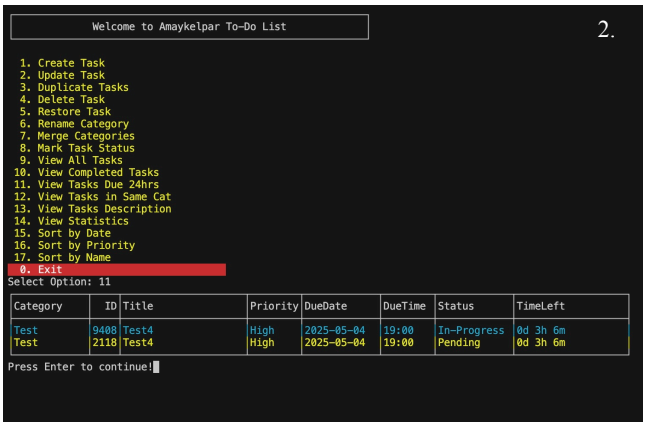
1. Status Tracking: Tasks can be marked as *"In-Progress" with blue color* or *"Complete" with green color* by using ANSI color definition.
2. Time Tracking: The system automatically calculates and displays time remaining until due dates by the current time.

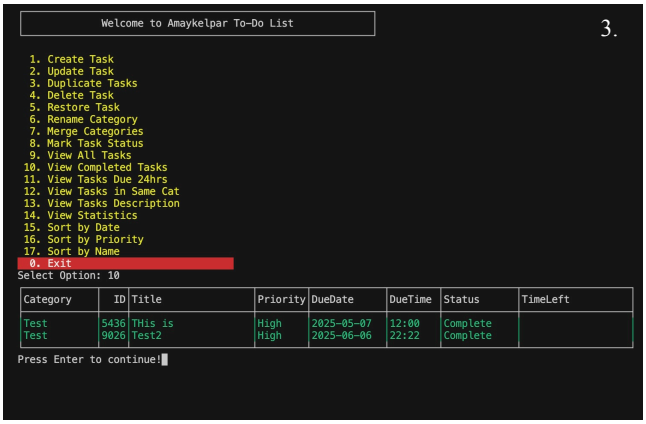


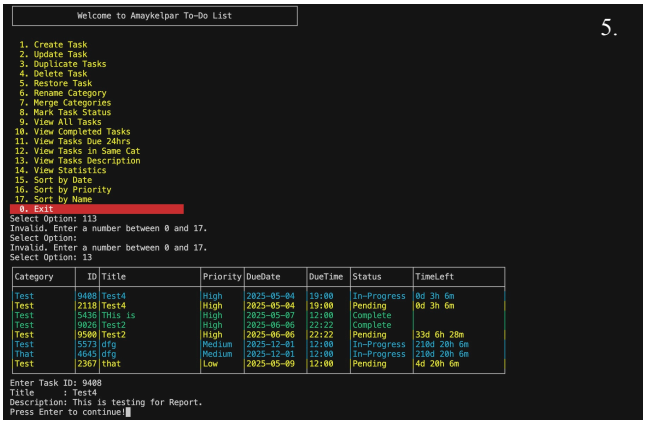
3.8.4. View Options

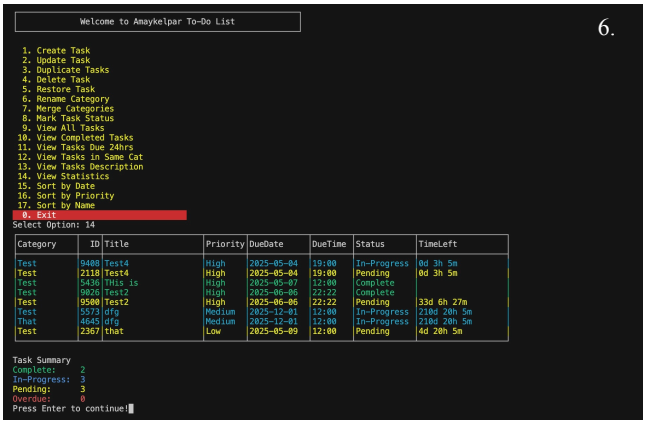
1. All Tasks View: Display all tasks in organized table.
2. Completed Tasks View: Show only completed tasks in the table.
3. Task due in 24 Hrs: Show tasks due within the next 24 hours from current time.
4. Category-specific View: View tasks that belong to a specific category.
5. Task Details View: View detailed descriptions of tasks.
6. Statistics View: Display statistics of all tasks status in summary.





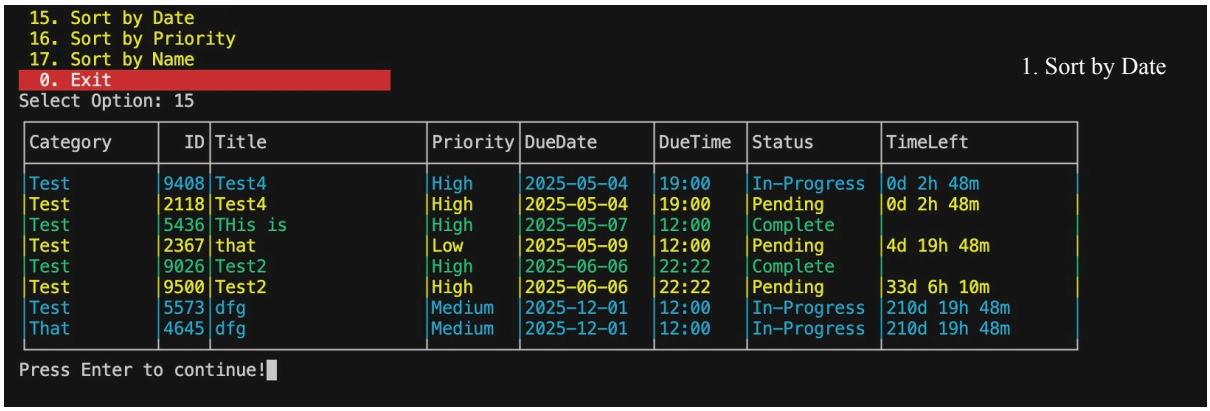


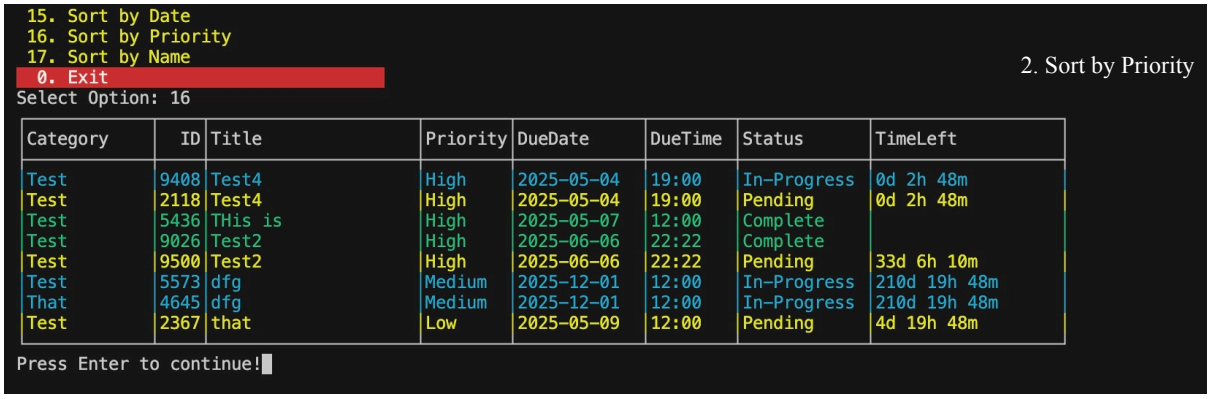


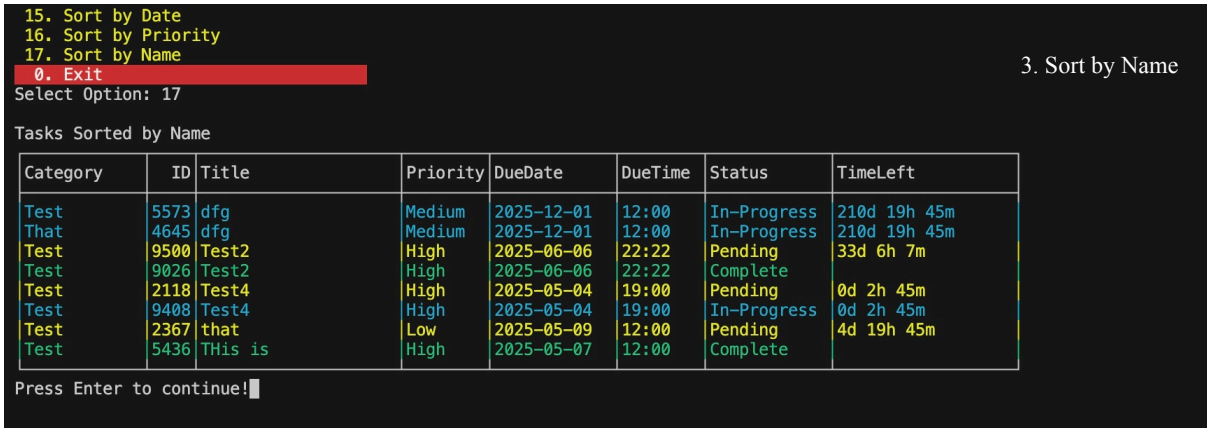


3.8.5. Sorting Options

1. Sort by Date: Sort tasks by their due dates.
2. Sort by Priority: Sort tasks by their priority levels.
3. Sort by Name: Sort tasks by their name. (Number first, then lowercase first and uppercase next).







3.8.6. Data Persistence

1. Save to CSV: Save all task data to a CSV file.
2. Load from CSV: Load task data from a CSV file on startup.

4. TIME COMPLEXITY AND TESTING

4.1 With Data Structures

Feature	Used Data Structure	Operation	Time Complexity
Create Task	Priority Queue (heap)	Inserting a task into priority queue	O(log n) (heap insert)
View All Tasks / View Today Tasks	Array + Sorting	Gathering tasks + nested bubble sort	$O(n^2)$
Sort by Priority	Priority Queue + Sorting	Flatten queue + sort with compareTask	$O(n^2)$
Sort by Date	Array + Date Comparator	Compare due dates/times in bubble sort	$O(n^2)$
Sort by Name	Circular Linked List	Insert tasks in order by name	$O(n^2)$ (linked insertion sort)
Delete Task	Stack (for undo)	Search by ID + stack push	O(n)
Restore Task	Stack + Queue insert	Stack pop + bubblein() into priority queue	O(log n)
Update Task	Priority Queue	Search by ID + direct update	O(n) (no index, linear search)
Mark Task Status	Priority Queue	Search and set status	O(n)
Statistics	Priority Queue	Scan all tasks and group by status	O(n)
Save/Load CSV	File I/O + Priority Queue	Serialize/deserialize tasks	O(n)

4.2 Without Data Structures

If our project were implemented using only plain arrays (no priority queue, no linked list, no stack), here's how the time complexity would be affected:

Feature	Plain Array Alternative	Time Complexity	Efficiency Impact
Create Task	Array append + sort	O(n) (if kept sorted) or O(1)append	May require manual sorting later
View Tasks / Sort	Full array + sort	$O(n^2)$ (bubble)	More work needed for each view
Delete Task (no stack)	Remove + shift	O(n)	No undo capability
Restore Task (manual reinsertion)	Manual re-entry	O(n)	No tracking of deleted tasks
Task Lookup	Linear search	O(n)	Slower search due to no indexing
Sort by Name	Array sort	$O(n^2)$	No reusable sorted structure

4.3 Summary Comparison Table

Feature	With Data Structures	Without Data Structures
Insertion (PriorityQueue)	O(log n)	O(1) or O(n)
Deletion (with Undo)	O(n) + O(1) stack push	O(n) (no undo)
Restoration	O(log n)	Manual and error-prone
Sorting (Bubble, Linked)	$O(n^2)$	O(n log n) (if optimized)
Searching	O(n)	O(n)

4.4 Testing and Edge Cases

The system was tested with variety of input and edge cases to ensure it works well:

1. Input Range: All numeric inputs (e.g., menu choices) are restricted to valid ranges.
2. Date & Time: Inputs must follow correct formats (YYYY-MM-DD, HH:MM) and are rejected if date is in the past or time is less than 30 minutes from the current time.
3. Negative Values: All inputs are validated to prevent negative or invalid input.
4. Task ID Handling: Functions like update, delete, and restore validate task IDs to avoid invalid operations.
5. Category Rules: Prevents renaming to an existing category or merging a category with itself.
6. Empty States: Handles cases such as no tasks, empty categories, or empty restore stack.
7. File Handling: Ensures missing or malformed CSV files do not crash the system.

4.5 Tasks and Schedule

Although we initially established a development schedule and divided roles among team members, the implementation phase was carried out collaboratively. We decided to work collectively on all components of the system to ensure consistency and prevent issues such as *syntax errors and function integration problems* that particularly from shared header files (.h). This collaborative approach allowed us to maintain code coherence and streamline debugging across modules.

Week	Aims	Aimed Outcomes
Week 1	Conduct research	Project plan Flowcharts & system diagrams
Week 2	Core Features Development	Task creation & deletion working Priority queue implemented Stack for restore implemented
Week 3	Advanced Features & Optimization	Circular Linked List to sort by name
Week 4	Testing	Bug fixes & performance improvements
Week 5	Documentation & Final Testing	Finalized project report

5. CONCLUSION

5.1 Conclusion

This project successfully demonstrates how data structures can be applied to solve real-life problems in a structured. By using priority queues, stacks, linked lists, and arrays, we developed a to-do list management system that supports efficient task organization, user interaction, and data persistence. Our system is reliable, extensible, and aligned with the objectives.

5.2 Challenges

During working on this project, we faced several challenges. Working with headers caused frequent syntax and linking issues due to shared functions. And in ensuring consistent data validation across all inputs (especially date and time) required a lot of works. Designing a system that allowed both task sorting and category grouping is complexity in structure and logic. And the last managing heap and ensuring correct priority behavior involved tons of debugging and testing.

5.3 Lessons Learned

From this project, we learned the importance of modular design, how to plan and implement data structures. We also learned how to apply stacks, priority queue, and linked lists. We also gained experience with memory management. Additionally, we understood how validation, error handling, and edge case planning are crucial for building reliable and user-friendly systems.