Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A Project Report on

"CLI To-Do List: Application of DSA"

[Code No: COMP 202]

(For partial fulfilment of I/II Year/Semester in Computer Engineering)

Submitted by

Dalton Khatri (Reg No: 037979-24)

Submitted to

Er. Sagar Acharya
Department of Computer Science and Engineering

Submission Date: 24/02/2026

Acknowledgement

I would like to express my sincere gratitude to everyone who contributed to the successful completion of this mini project, "CLI To-Do List: Application of DSA".

First and foremost, I extend my deepest thanks to my project supervisor, Er. Sagar Acharya, from the Department of Computer Science and Engineering, Kathmandu University. His guidance on data structures and algorithm design was invaluable throughout the development of this project. His encouragement to explore the practical applications of fundamental data structures helped me gain a deeper understanding of linked lists, stack-based memory management, heap operations, and queue-based scheduling.

I am also thankful to the Department of Computer Science and Engineering, Kathmandu University, for providing the necessary resources and a supportive learning environment that made it possible to complete this project successfully.

Finally, I would like to thank my family and friends for their constant encouragement and motivation throughout the duration of this project.

Dalton Khatri
Regd. No. 037979-24
24th February 2026

Abstract

This project presents the design and implementation of a Command-Line Interface (CLI) based To-Do List application in C++ demonstrating the practical application of four fundamental data structures within a single real-world system. A Singly Linked List serves as the master task storage, supporting dynamic insertion, deletion, traversal, and keyword-based search. A Stack implemented via linked list provides LIFO-based undo functionality for deleted tasks in O(1) time. A Min-Heap based Priority Queue assigns numeric priorities from P1 to P10 with a cascade push-down mechanism that automatically shifts existing tasks on collision, surfacing the most urgent task in O(1) via heap peek. A circular array-based Queue implements a Daily Planner where tasks are processed in FIFO order with options to mark done or skip. The entire application is built in standard C++17 with no external libraries, compiled with zero warnings, and demonstrates how multiple data structures can coexist and complement each other within a single cohesive application.

Keywords: Singly Linked List, Stack, Min-Heap, Priority Queue, Circular Queue, C++, Data Structures, CLI Application

Table of Contents

List of Figures

List of Tables

Abbreviations

CLI Command-Line Interface

DSA Data Structures and Algorithms

LIFO Last In First Out

FIFO First In First Out

C++ C Plus Plus (Programming Language)

I/O Input / Output

ID Identifier

O(n) Order of n (Big-O Notation)

O(1) Order of Constant Time

O(log n) Order of Logarithmic Time

Chapter 1: Introduction

1.1 Background

Task management is one of the most fundamental challenges in both academic and professional environments. As the volume of responsibilities grows, the need for a structured and efficient system to organize, prioritize, and track tasks becomes increasingly important. Traditional approaches such as paper-based lists or basic digital notes lack the ability to enforce priority ordering, recover from mistakes, or guide users through a planned sequence of work. This project addresses these limitations by building a CLI-based To-Do List application in C++ that is backed by well-defined data structures rather than high-level abstractions or database systems. The application is built entirely from scratch using a Singly Linked List, Stack, Min-Heap, and Queue, each chosen deliberately for the specific operations they optimize. By grounding a practical everyday tool in fundamental data structures, this project bridges the gap between theoretical DSA concepts and their real-world utility.

1.2 Objectives

1. To implement a Singly Linked List as the primary storage mechanism for task data, supporting dynamic insertion, deletion, traversal, and keyword-based search operations.
2. To implement a Stack using a linked list to provide an undo mechanism for deleted tasks, demonstrating the LIFO principle in a practical context.
3. To implement a Min-Heap-based Priority Queue that assigns numeric priorities from P1 to P10 to each task, with a cascade push-down mechanism to resolve priority collisions automatically.
4. To implement a circular array-based FIFO Queue as a Daily Planner, enabling users to schedule and work through tasks sequentially within a single session.

5. To develop a fully functional, menu-driven CLI application in standard C++17 that integrates all four data structures cohesively without the use of any external libraries.

1.3 Motivation and Significance

The motivation behind this project stems from the observation that data structures are often taught in isolation through abstract examples such as sorting numbers or traversing trees, which can make it difficult for students to appreciate their relevance in practice. By embedding four distinct data structures into a single application where each one serves a uniquely justified purpose, this project demonstrates that the choice of data structure is not academic but consequential; it directly determines what operations are efficient and what features are even possible. The significance of this work lies in its ability to show, concretely, why a Min-Heap is the right tool for priority-based retrieval, why a Stack naturally models undo behaviour, why a Queue captures the sequential nature of daily planning, and why a Linked List suits dynamic task storage better than a fixed array. For a student of Data Structures and Algorithms, this project serves as a practical reference for how these structures translate from textbook definitions into working, useful software.

Chapter 2: Related Works

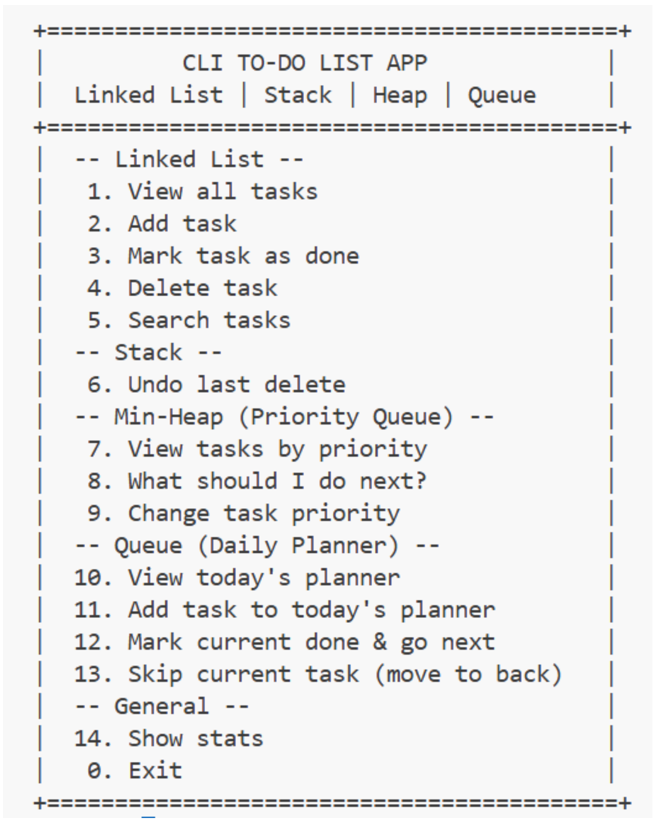
Early task management systems transitioned from paper-based lists to digital applications that stored tasks in fixed-size arrays, which required costly resizing and shifting for dynamic operations. This limitation established the case for linked list-based storage. Cormen et al. in Introduction to Algorithms provide the theoretical foundation for heap operations, confirming that a Min-Heap guarantees O(log n) insertion and O(1) minimum retrieval, while queue-based FIFO scheduling has been extensively applied in operating system process management and network routing both directly informing the data structure choices in this project.

Existing CLI task managers such as Taskwarrior, Todo.txt, and topydo offer priority support and tagging but depend on external libraries and file I/O abstractions that obscure the underlying data structure decisions. Stack-based undo is similarly well established across tools like Microsoft Word and Visual Studio Code, where a command stack records each action for reversal. This project distinguishes itself by implementing every component storage, undo, priority ranking, and daily planning from scratch in standard C++17, keeping all internal mechanics transparent and academically demonstrable while extending the standard priority model with an original cascade push-down mechanism that enforces strict priority uniqueness across all pending tasks.

Chapter 3: Methodology

3.1 System Overview

The CLI To-Do List application is designed as a single-file C++17 program that integrates four independently implemented data structures Singly Linked List, Stack, Min-Heap, and Queue into one cohesive system. Each structure is implemented from scratch using classes and raw pointers with manual memory management, without relying on any standard library containers such as std::vector, std::stack, or std::priority queue. The application exposes all functionality through a numbered menu loop in main(), and all four structures are coordinated within a single TodoList class that ensures their states always remain synchronized. For example, when a task is marked as done, it is simultaneously removed from the Min-Heap and dequeued from the Daily Planner if present, preventing any inconsistency between the structures.



```
+==========================================+
|          CLI TO-DO LIST APP              |
|  Linked List | Stack | Heap | Queue      |
+==========================================+
|  -- Linked List --                       |
|   1. View all tasks                      |
|   2. Add task                            |
|   3. Mark task as done                   |
|   4. Delete task                         |
|   5. Search tasks                        |
|  -- Stack --                             |
|   6. Undo last delete                    |
|  -- Min-Heap (Priority Queue) --         |
|   7. View tasks by priority              |
|   8. What should I do next?              |
|   9. Change task priority                |
|  -- Queue (Daily Planner) --             |
|  10. View today's planner                |
|  11. Add task to today's planner         |
|  12. Mark current done & go next         |
|  13. Skip current task (move to back)    |
|  -- General --                           |
|  14. Show stats                          |
|   0. Exit                                |
+==========================================+
```

Figure 1: Main Menu of the CLI To-Do List Application

3.2 Singly Linked List Task Storage

The primary data structure of the application is a Singly Linked List, chosen for its ability to support dynamic insertion and deletion without requiring pre-allocated memory. Each node in the list is a TaskNode struct containing an integer ID, a string title, a boolean completion flag, an integer priority value, a boolean daily planner flag, a string timestamp, and a pointer to the next node. Tasks are always appended to the tail of the list, achieved by traversing to the last node and linking the new node, giving O(n) tail insertion in the absence of a tail pointer a deliberate choice to keep the implementation transparent and educational. Deletion requires a linear scan to locate the target node and pointer adjustment to bypass it, also O(n). Traversal for display and search is similarly O(n). The linked list was selected over an array because the number of tasks is not known in advance, and frequent insertions and deletions would require costly shifting in a fixed array. All node memory is allocated on the heap using new and freed explicitly in the destructor, demonstrating manual memory management in C++.

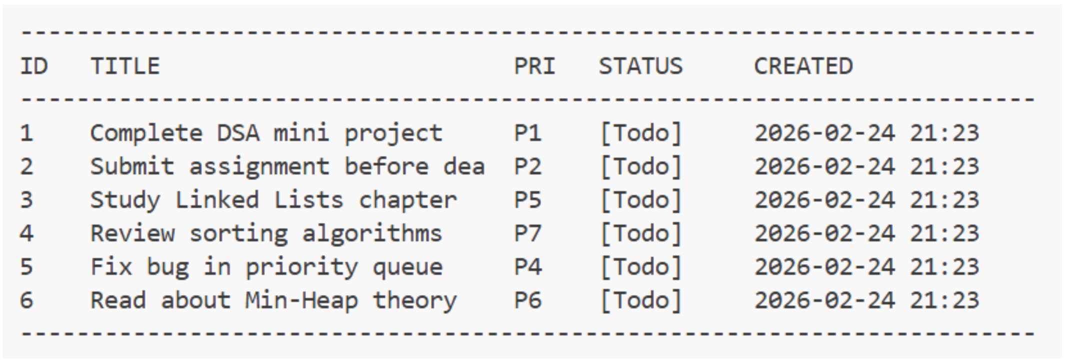
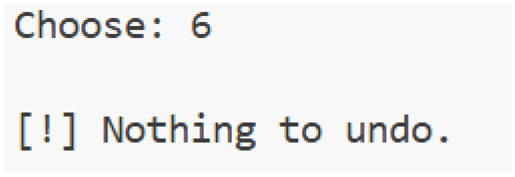

```
---------------------------------------------------------------------
ID   TITLE                         PRI   STATUS     CREATED
---------------------------------------------------------------------
1    Complete DSA mini project     P1    [Todo]     2026-02-24 21:23
2    Submit assignment before dea  P2    [Todo]     2026-02-24 21:23
3    Study Linked Lists chapter    P5    [Todo]     2026-02-24 21:23
4    Review sorting algorithms     P7    [Todo]     2026-02-24 21:23
5    Fix bug in priority queue     P4    [Todo]     2026-02-24 21:23
6    Read about Min-Heap theory    P6    [Todo]     2026-02-24 21:23
---------------------------------------------------------------------
```

Figure 2: Task List Display with Priority and Status

3.3 Stack Undo Mechanism

The undo functionality is implemented using a Stack built from a secondary linked list of StackNode elements, each holding a pointer to a TaskNode copy and a pointer to the next StackNode. When a task is deleted from the main linked list, a deep copy of that task is immediately pushed onto the stack before the original node is freed. This ensures the undo stack holds its own independently allocated copies, eliminating any dangling pointer risk. When the user triggers undo, the top of the stack is popped, and the recovered task is re-appended to the tail of the main linked list. Both push and pop operations run in O(1) time, making undo instantaneous regardless of how many tasks are in the list. The stack follows the Last In First Out (LIFO) principle, meaning the most recently deleted task is always the one recovered, consistent with the behaviour users expect from undo in any software system. The stack is also automatically drained in its destructor to prevent memory leaks.



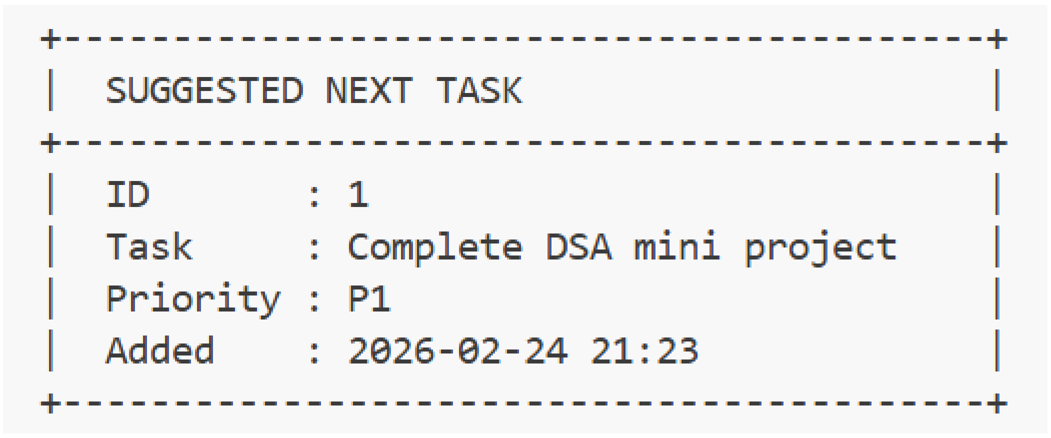
```
Choose: 6

[!] Nothing to undo.
```

Figure 3: Undo Delete Restoring a Task

3.4 Min-Heap Priority Queue

Priority management is handled by a Min-Heap, implemented as a fixed-size array of HeapEntry structs, each containing an integer priority and an integer task ID. The heap maintains the property that the entry with the smallest priority number always sits at the root (index 0), making the highest-urgency task retrievable in O(1) via peekMin(). Insertion places the new entry at the end of the array and calls heapifyUp(), which repeatedly swaps the entry with its parent while it is smaller, restoring the heap property in O(log n). Deletion by task ID requires a linear O(n) scan to locate the entry, after which it is replaced by the last entry, and heapifyDown() is called to restore the heap property downward in O(log n). Priority updates follow the same pattern locate in O(n), then call both heapifyUp and heapifyDown to handle both directions of movement.

A key feature of the priority system is the cascade push-down algorithm, which runs before every task insertion and priority change. The priority range is P1 (most urgent) to P10 (least urgent), and each pending task must occupy a unique priority slot. When a user assigns a priority that is already taken, the system calls cascadeDown(), which first checks whether the destination slot (priority+1) is also occupied before moving the current occupant. This deepest-first recursive approach ensures the destination is always free before any task is moved into it, preventing any task from being overwritten. The heap is also used to generate a sorted priority snapshot for the "view by priority" feature, achieved by copying the heap array and applying a simple selection sort on the copy without disturbing the original heap structure.

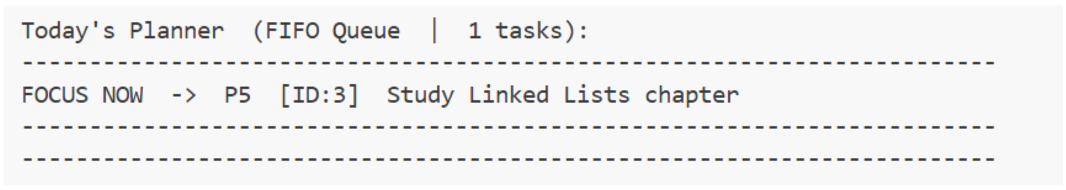


```
+----------------------------------------+
|  SUGGESTED NEXT TASK                   |
+----------------------------------------+
|  ID       : 1                          |
|  Task     : Complete DSA mini project  |
|  Priority : P1                         |
|  Added    : 2026-02-24 21:23           |
+----------------------------------------+
```

Figure 4: Suggested Next Task via Heap Peek

3.5 Queue Daily Planner

The Daily Planner feature is implemented using a circular array-based Queue with front and rear integer pointers and a count variable. The circular design allows the array to wrap around using modular arithmetic, enabling O(1) enqueue and O(1) dequeue without ever shifting elements. The queue stores task IDs only, and always retrieves the full task details by looking up the ID in the linked list. The front of the queue at any time represents the current task the user should focus on. Users can either mark the front task as done which dequeues it, marks it complete in the linked list, and removes it from the heap or skip it, which dequeues it from the front and re-enqueues it at the rear, effectively moving it to the back of today's plan. A contains() method prevents the same task from being added to the planner twice, and a remove() method allows a specific task ID to be purged from anywhere in the queue when needed, such as when a planned task is deleted from the main list.



```
Today's Planner  (FIFO Queue  |  1 tasks):
-----------------------------------------------------------------------
FOCUS NOW  ->  P5  [ID:3]  Study Linked Lists chapter
-----------------------------------------------------------------------
-----------------------------------------------------------------------
```

Figure 5: Daily Planner Queue Display

3.6 Inter-Structure Synchronization

A critical aspect of the system design is keeping all four structures in sync. The TodoList class acts as the coordinator, and every public operation that modifies one structure also updates the others where necessary. The table below summarizes which structures are affected by each major operation:

Operation	**Linked List**	**Stack**	**Min-Heap**	**Queue**
Add task	Insert Node	-	Insert Entry	-
Delete task	Remove Node	Push Copy	Remove Entry	Remove if present
Mark Done	Update Flag	-	Remove Entry	Remove if present
Undo Delete	Re-append node	Pop Copy	Re-Insert Entry	-
Add to Planner	Update flag	-	-	Enqueue ID
Planner Done	Update flag	-	Remove Enrtry	Dequeue
Change Priority	Update field	-	Update Entry	-

Table 1: Inter-Structure Synchronization

This coordination ensures that no structure ever holds stale or inconsistent data, and the application behaves correctly regardless of the order in which operations are performed.

3.7 Algorithm Complexity Summary

Operations	Data Structure	Time Complexity
Add Task	Linked List + Min-Heap	O(n) cascade + O(log n) heap insert
Delete Task	Linked List + Min-Heap	O(n) scan + O(log n) heap remove
Undo Task	Stack + Linked List	O(1) pop + O(n) append
Suggest next	Min-Heap	O(1) peek
View by Priority	Min-Heap snapshot	O(n log n) sort on copy
Change Priority	Min-Heap	O(n) scan + O(log n) rebalance
Add to Planner	Queue	O(1) enqueue
Planner done/skip	Queue + Heap	O(1) dequeue + O(log n) heap remove
Search	Linked List	O(n * m) where m = keyword length

Table 2: Algorithm Complexity Summary

Chapter 4: Discussion on Achievement

This chapter presents the results obtained from testing the CLI To-Do List application, reflects on the achievements of the implementation, discusses the features delivered, and describes the key challenges encountered during development along with how they were resolved.

4.1 Achievement Overview

The project successfully delivered a fully functional CLI-based task management application built entirely in standard C++17 without any external libraries. All fourteen menu operations were implemented and tested, confirming correct behaviour across all four data structures. The Singly Linked List correctly handles dynamic task insertion, deletion, and traversal. The Stack reliably restores deleted tasks in LIFO order with no memory leaks or dangling pointer issues. The Min-Heap consistently surfaces the highest-priority pending task in O(1) and correctly rebalances after every insertion, deletion, and priority update. The cascade push-down algorithm was verified to correctly resolve multi-level priority collisions without overwriting any existing task. The circular Queue correctly enqueues and dequeues tasks in FIFO order, and the skip operation accurately moves the front task to the rear. All four structures remain synchronized throughout every operation, meaning no inconsistency was observed between the linked list, heap, and queue states during testing.

4.2 Features Implemented

The following features were successfully implemented and verified in the final application:

- Linked list-based dynamic task storage with unique auto-incremented IDs and creation timestamps
- Task addition with numeric priority assignment (P1 to P10) and automatic cascade push-down on collision
- Task deletion with stack-based undo supporting multi-level restoration

- Min-Heap priority queue with O(1) suggestion of the next most urgent task
- Priority view that generates a sorted snapshot of pending tasks without modifying the heap
- Priority change with collision handling via cascade push-down and temporary task parking
- Circular array-based Daily Planner Queue with enqueue, dequeue, mark-done, and skip operations
- Keyword-based linear search across all tasks
- Statistics display showing total, completed, pending, heap size, and planner queue size
- Full inter-structure synchronization ensuring all structures always reflect consistent state.

4.3 Challenges and Difficulties

The most significant challenge encountered during development was implementing the cascade push-down algorithm correctly. The initial approach moved a task from its current priority slot to the next slot and then recursively called cascadeDown on the new slot. However, this caused the same task to keep chasing itself down the priority range, eventually reaching P10 regardless of whether any other task was present. The fix required inverting the recursion order checking and clearing the destination slot first before moving the current task into it, a deepest-first approach that guaranteed the destination was always free before any move occurred. This required careful reasoning about pointer aliasing and recursive state, and was resolved through step-by-step tracing of the call stack with sample data.

A second challenge was maintaining synchronization between the four data structures. Early versions of the application updated only the linked list during operations like mark-done and delete, leaving stale entries in the heap and planner queue. This was resolved by centralizing all operations inside the TodoList class and explicitly updating every affected structure within each method, making cross-structure consistency a design invariant rather than an afterthought. Memory management also required careful attention ensuring that deep copies were made before deletion (for the undo stack), and that all allocated nodes were properly freed in destructors which was verified by tracing all new and delete calls throughout the codebase

Chapter 4: Conclusion

This project successfully demonstrates the integration of four fundamental data structures.Singly Linked List, Stack, Min-Heap, and Queue into a single functional CLI application that solves a genuine everyday problem. Each data structure was selected based on the time complexity requirements of the feature it supports: the linked list for dynamic storage, the stack for O(1) undo, the min-heap for O(1) priority retrieval with O(log n) updates, and the circular queue for O(1) FIFO daily planning. The cascade e push-down mechanism for priority collision resolution represents an original design contribution that extends the standard heap model to enforce strict priority uniqueness. The project was implemented entirely in standard C++17 with manual memory management and no external dependencies, compiling with zero warnings and demonstrating correct behaviour across all fourteen menu operations.

However, the application has several limitations that present opportunities for future work. Task data is not persisted between sessions all tasks are lost when the program exits, as there is no file I/O implementation. The search functionality is case-sensitive, meaning a search for "dsa" will not match "DSA". Task IDs are never reused after deletion, which means the ID counter increments indefinitely and gaps appear in the ID sequence over time. The priority system is capped at P10, which limits the number of concurrently pending tasks that can each hold a unique priority to ten. The application is single-user and single-session with no support for task categories, due dates, or recurring tasks. Future enhancements could address these limitations by adding file-based persistence using standard file I/O, implementing case-insensitive search, introducing a doubly linked list for bidirectional traversal, extending the priority range, and adding a due date field with a secondary sort key in the heap for tie-breaking between tasks of equal priority.

Chapter 5: References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

Taskwarrior. (2024). *Task Management for the Command Line*. Retrieved from https://taskwarrior.org

Todo.txt. (2024). *Todo.txt: Future-proof task tracking in a file you control*. Retrieved from https://todotxt.org

Stroustrup, B. (2013). *The C++ Programming Language* (4th ed.). Addison-Wesley.