Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A Project Report

on

"Contact Management System"

[Code No: COMP 202]

For partial fulfillment of Second Year/first Semester in Computer Engineering

Submitted By:

Manish Khatri [25]

Submitted To:

Mr. Sagar Acharya

Department of Computer Science and Engineering

Submission Date: 2026-02-25

Bona fide Certificate

This is to certify that the project entitled

" Contact Management System "

is a bonafide work carried out by the student in partial fulfillment of the requirements for the Second Year /First Semester of the Bachelor of Engineering in Computer Engineering, under the guidance of Mr. Sagar Acharya, Department of Computer Science and Engineering.

Name of Student :

Manish Khatri

The project work has been carried out with sincerity, dedication, and to the satisfaction of the department, adhering to the academic guidelines.

Project Supervisor

_____

Mr. Sagar Acharya

Department of Computer Science and Engineering

Date: 2026-02-25

Acknowledgement

I would like to express my sincere and heartfelt gratitude to my respected teacher, **Mr. Sagar Acharya**, for his continuous guidance, invaluable suggestions, and constant encouragement in the field of Data Structures and Algorithms. His excellent teaching, insightful explanations, and dedication to the subject inspired me to undertake and successfully complete this project. His support played a vital role in strengthening my understanding of the core concepts that formed the foundation of this work.

I would also like to extend my sincere appreciation to the Department of Computer Science and Engineering for providing me with the opportunity and platform to apply my theoretical knowledge in a practical and meaningful way. The academic environment, resources, and encouragement provided by the department were instrumental in helping me carry out and complete this project successfully.

Finally, I would like to express my gratitude to everyone who directly or indirectly supported and encouraged me during the course of this project. Their motivation and support contributed significantly to the successful completion of this work. I am truly thankful for their assistance and encouragement.

Thank You

Abstract

Contact management is a common real-world application that requires efficient storage, retrieval, and manipulation of personal information. This project presents a Contact Management System developed in C++ with a console-based interface. The primary objective is to demonstrate the practical use of fundamental data structures – Linked List, Stack, and Queue – in a cohesive and interactive manner. The system stores contacts (name, phone, email) in a Singly Linked List, allowing dynamic memory allocation and easy insertion, deletion, and traversal. A Stack is implemented to remember the last five actions (add, update, delete), providing a basic undo-like functionality. A Queue (circular array) keeps track of the five most recently viewed contacts, offering quick access to frequently accessed entries. Optionally, a Binary Search Tree can be enabled to enable faster searching by name, demonstrating hierarchical data organization.

The console interface presents a clear menu with options to add, display, search, update, delete, view recent contacts, and show the undo stack. All operations update the structures in real time, and the user sees immediate feedback. The project effectively illustrates how multiple data structures can work together in a single application, bridging the gap between theoretical concepts and practical implementation.

Keywords: Contact Management, C++, Linked List, Stack, Queue, Binary Search Tree, Data Structures, Console Application.

Table of Contents

List of Figures

Acronyms/Abbreviations

The list of all abbreviations used in the documentation are as follows :

DSA: Data Structures and Algorithms

BST: Binary Search Tree

LIFO: Last In, First Out

FIFO: First In, First Out

OOP: Object-Oriented Programming

IDE: Integrated Development Environment

UI: User Interface

CPP: C++ Programming Language

LL: Linked List

QDS: Queue Data Structure

SDS: Stack Data Structure

Chapter 1 Introduction

1.1 Background

Data Structures and Algorithms form the foundation of computer science, providing the essential principles needed to build efficient and organized software solutions. Understanding how different data structures work internally—and how they can be combined to solve real problems—is crucial for students who want to become good programmers. However, traditional ways of teaching DSA often focus on theory and isolated examples, where each data structure is taught separately without showing how they can work together in a single application.

Many students struggle to see the practical connections between different structures. For example, they learn that a stack follows LIFO (Last In, First Out) and a queue follows FIFO (First In, First Out), but they rarely get to see how both can be useful in the same program. Similarly, they understand that a linked list allows dynamic memory allocation, but they may not realize how it can serve as the core storage while other structures handle additional features like undo history or recently accessed items. Without a project that brings multiple structures together, these concepts remain isolated in the mind.

To address this gap, this project presents a **Contact Management System** developed in **C++** that integrates three fundamental data structures in one application. The **Singly Linked List** stores all contact information, allowing unlimited growth as new contacts are added. A **Stack** (implemented as a fixed-size array) remembers the last five actions performed by the user, providing a simple way to track recent changes. A **Queue** (circular array) keeps a record of the five most recently viewed contacts, demonstrating how FIFO behavior can be used to maintain access history. Optionally, a **Binary Search Tree** can be enabled to show how searching can be optimized when data is organized hierarchically.

Unlike textbook examples that demonstrate each structure in isolation, this system shows them working together in a familiar context. Users can add, search, update, and delete contacts while the stack and queue automatically update in the background. This integration helps students understand not just what each structure does, but how they can collaborate to create a more feature-rich application. The console interface keeps everything simple and transparent, allowing learners to focus on the behavior of the structures rather than complex graphics.

1.2 Objectives

The primary objectives of the DSA Visualizer Project are as follows:

- To implement a **Singly Linked List** that stores contact information (name, phone, email) and supports insertion, display, search, update, and deletion.

- To use a **Stack** (array-based) to remember the last five user actions, providing an undo-like functionality.

- To use a **Queue** (circular array) to keep track of the five most recently viewed contacts.

- To create a simple, menu-driven console interface that lets users interact with all these structures.

- To serve as an educational tool that helps other students understand how multiple data structures can work together in a single application.

1.3 Motivation and Significance

The motivation behind this project comes from the need to understand how multiple data structures can work together in a single application rather than studying them in isolation. In traditional learning, we often implement stacks, queues, and linked lists separately through small textbook exercises. While this helps understand individual concepts, it does not show how these structures can collaborate to build something useful. This project addresses that gap by creating a Contact Management System where a linked list stores contacts, a stack remembers recent actions, and a queue tracks viewed contacts—all working together in one program.

This project is significant because it demonstrates the practical integration of data structures in a meaningful context rather than using them as standalone examples. The linked list shows how dynamic memory allocation allows the contact list to grow and shrink as needed. The stack illustrates LIFO behavior by storing the last five actions (add, update, delete) in order, letting users see their recent activity. The queue demonstrates FIFO principle by keeping track of the five most recently viewed contacts, showing how a circular buffer can maintain access history. Together, these structures provide a complete picture of how different data organization methods can serve different purposes within the same application.

Furthermore, the simple console-based interface keeps the focus on the behavior of the data structures themselves without the distraction of complex graphics. Users can clearly see how each operation affects the linked list, how actions pile up on the stack, and how the queue maintains its order. The project not only strengthens understanding of pointers, dynamic memory, and algorithm design but also serves as a learning tool for other students who want to see data structures in action. Its clear code organization, modular structure, and practical functionality make it suitable for academic evaluation, viva examinations, and as a foundation for future enhancements.

Chapter 2 Related Works

Several existing applications and educational resources have explored contact management systems and data structure implementations. Many simple contact management programs are available online as tutorial examples, typically implemented using arrays or linked lists in C++. Websites like GeeksforGeeks and Tutorialspoint provide numerous code samples that demonstrate basic operations like adding, searching, and deleting contacts using a singly linked list. These examples are helpful for understanding the core concepts but usually focus on only one data structure and do not integrate additional features like undo history or recently viewed items.

Traditional implementations of data structures are commonly taught using console-based C++ programs in academic courses. Standard textbooks such as **Introduction to Algorithms** by Cormen et al. and **Data Structures, Algorithms, and Applications in C++** by Sahni provide comprehensive theoretical explanations of linked lists, stacks, and queues. However, these books typically present each data structure in separate chapters with isolated examples. While they build strong theoretical foundations, they rarely show how multiple structures can be combined in a single practical application.

Several student projects and open-source repositories on GitHub demonstrate contact management systems with varying levels of complexity. Some projects use file handling to store contacts permanently, while others implement sorting and searching features. However, most of these projects rely on a single data structure—usually an array or linked list—and do not incorporate auxiliary structures like stacks for undo functionality or queues for tracking recent activity. A few more advanced projects include features like binary search trees for faster lookup, but they often lack the integration of multiple structures working together seamlessly.

Mobile and desktop contact manager applications like Google Contacts, Microsoft Outlook, and open-source alternatives provide rich features but are designed for end-users rather than educational purposes. Their internal implementations are complex and not accessible for learning how data structures operate. While these applications demonstrate what a fully-featured contact manager can do, they do not help students understand the underlying data structure concepts.

This project addresses the gap by creating a simple yet complete contact management system that explicitly integrates **three different data structures** in one application. The linked list serves as the primary storage, the stack maintains action history, and the queue

tracks recently viewed contacts. By keeping the implementation transparent and the interface simple, this project allows students to see exactly how each structure contributes to the overall functionality. It bridges the gap between textbook examples and real-world applications, showing how multiple data structures can work together in a way that is both educational and practical.

Chapter 3 Design and Implementation

3.1 Implementation

The system was developed using an object-oriented approach in **C++**, with separate header files for each data structure and a modular file organization.
The project is divided into folders: structures/ (contains all data structure classes), ui/ (menu handling), utils/ (helper functions like clear screen, pause), and the main driver file.

3.1.1 Implementation Planning

The project was planned around four core data structures:

- **Singly Linked List:** Stores all contacts dynamically. Each node contains a Contact structure (name, phone, email) and a pointer to the next node.
- **Stack (array-based):** Holds the last five actions as strings (e.g., "Added Ram", "Updated Sita"). It follows LIFO order.
- **Queue (circular array):** Stores the names of the last five viewed contacts. It follows FIFO order.
- **Binary Search Tree (optional):** Stores contacts in a tree structure keyed by name for faster searching.

During planning, I decided to keep the console interface simple so that users can focus on the data structure operations. All actions are triggered from a main menu, and each operation updates the relevant structures immediately.

3.1.2 Requirement Analysis

Functional Requirements

- The system must allow users to add a new contact (name, phone, email).
- It must display all contacts in the linked list.
- Users must be able to search for a contact by name (using linked list traversal; if BST is enabled, search becomes faster).
- Contact details (phone, email) must be updatable.
- Contacts must be deletable by name.
- The last five actions (add, update, delete) must be stored in a stack and displayed on request.
- The last five viewed contacts (via search or update) must be stored in a queue and displayed on request.
- All operations must provide appropriate feedback (success/failure messages).

Non-Functional Requirements

- The program must handle invalid input gracefully (e.g., non-numeric menu choices).
- It should be portable and compile on any system with a C++ compiler.
- The code must be well-commented and modular for easy understanding.

3.1.3 System Design

The system follows a modular design:

- Contact.h defines the Contact structure.
- LinkedList.h contains the LinkedList class with methods: addContact, displayAll, searchByName, updateContact, deleteByName.
- Stack.h contains the Stack class with push, pop, and display methods.
- Queue.h contains the Queue class with enqueue, and displayRecent methods.
- BST.h (optional) contains the BST class with insert, search, and inorder traversal.
- Menu.h / Menu.cpp handle the menu display and user input.
- helpers.h provides clearScreen() and pause() functions.
- main.cpp integrates everything and runs the main loop.

The global instances of LinkedList, Stack, and Queue are declared in main.cpp and passed to the menu functions as needed (or used directly via extern). This keeps the design simple.

Data Structure Implementations

Stack Implementation:

A fixed-size array (size 5) stores C-strings. push adds a new action; if the stack is full, it shifts all elements to remove the oldest (a simple approach to keep only the last five). pop retrieves the most recent action.

Queue Implementation:

A circular array of size 5 stores names. enqueue adds a name; if the queue is full, it overwrites the oldest. displayRecent prints the names in order from oldest to newest.

Linked List Implementation:

Each node is dynamically allocated. addContact appends at the end. searchByName traverses from head. deleteByName handles removal from head or middle. The destructor frees all nodes.

Binary Tree Implementation:

Nodes contain a Contact and left/right pointers. Insertion follows BST rules. Search uses recursion. In-order traversal displays contacts in sorted order..

UI Components

The UI module provides a simple console-based menu system with numbered options and user-friendly prompts. Helper functions like clearScreen() handle screen clearing after each operation, and pause() waits for user input before continuing, creating a clean and organized flow. Input validation using cin.fail() checks for invalid entries and prevents crashes when users make mistakes. The menu is displayed inside a bordered box created using characters to make it visually distinct and easy to read. These small touches make the console application feel more polished despite the lack of graphical elements.

Mode Management

The application uses a menu-driven approach where each user choice directly calls the corresponding function rather than using an explicit mode variable. When a contact is added, not only does the linked list get updated, but the action is automatically pushed onto the stack and the contact name is enqueued in the recent queue. Similarly, search and update operations automatically add the contact name to the recent queue without requiring the user to switch modes. This design shows how different data structures can work together seamlessly while the user simply focuses on managing contacts.

Data Structure Integration

Linked List: The singly linked list serves as primary storage with nodes containing name, phone, email, and pointer to next node, demonstrating dynamic memory allocation in a practical context. Stack: The fixed-size array stack stores the last five actions as strings, showing LIFO behavior as new actions are pushed and old ones are discarded when full. Queue: The circular array queue stores the last five viewed contact names, demonstrating FIFO behavior as new names are enqueued and oldest are overwritten. This structured implementation ensures that each data structure is not only functionally correct but also educationally visible, providing an interactive demonstration of fundamental DSA concepts in a real-world application.

3.1.4 Development

The development of the project was carried out in phased stages to ensure modularity, correctness, and stability. Initially, the core data structure classes were implemented and tested with basic operations in isolation. This phase validated the correct functioning of Linked List add/display/search/update/delete, Stack push/pop for action history, Queue enqueue/display for recent contacts, and optional BST insertion before integrating everything into the main program.

Subsequently, the console menu interface was developed around the validated core logic. The menu functions were implemented incrementally, starting with simple option display and progressing to full integration with all data structure operations. The helper functions like clearScreen() and pause() were added to provide smooth navigation and better user experience during program execution.

The major development phases of the project can be summarized as follows:

1. Core class design: Contact struct, LinkedList, Stack, Queue, BST (optional)
2. Console validation: Testing all operations for correctness in isolation
3. Basic menu setup: Main loop with numbered options and switch case
4. UI development: Clear screen and pause functions for clean navigation
5. Integration implementation: Connecting menu choices to structure operations
6. Stack integration: Adding action tracking for add/update/delete operations
7. Queue integration: Adding recent contacts tracking for search/update
8. Testing and refinement: Ensuring all structures work together without errors
9. Documentation: Adding comments and organizing code into separate files

This structured development approach ensured that the project was robust, interactive, and educational, successfully integrating core DSA concepts with a functional console interface.

3.1.5 Testing and Debugging

The project underwent testing to ensure stability, correctness, and smooth user interaction. Initial testing focused on data structure operations in isolation, verifying that Linked List add/display/search/update/delete correctly managed node connections, Stack push/pop maintained LIFO order for action history, Queue enqueue/display preserved FIFO behavior for recent contacts, and optional BST insertions followed binary search tree rules.

Integration testing verified that menu options correctly triggered the appropriate structure operations and that all structures updated accurately after each action. The stack was tested to ensure it only kept the last five actions and discarded older ones properly. The queue was tested to confirm that it maintained the correct order of recent contacts and overwrote the oldest when full. The linked list was tested with various sequences of adds and deletes to ensure pointers were always correctly maintained.

Memory management was carefully handled, particularly for the linked list where each new node uses dynamic memory allocation. The destructor was tested to ensure all nodes are properly deleted when the program ends, preventing memory leaks. Edge cases like deleting the head node, deleting from an empty list, and searching for non-existent contacts were all tested to ensure the program handled them gracefully without crashing.

Through iterative testing and refinement, the system now performs reliably, providing a crash-free, interactive, and user-friendly experience that demonstrates the intended Linked List, Stack, Queue, and optional Binary Search Tree concepts working together in a single application.

3.2 System Requirement Specifications

The system was developed and tested with specific software and hardware requirements to ensure proper functionality, performance, and compatibility. The specifications are detailed below.

3.2.1 Software Specifications

The following software components are required for the development and execution of the system:

- **Programming Language:** C++11 or later
- **Compiler:** MinGW (Windows), GCC (Linux/Mac), or any standard C++ compiler
- **Build Tools:** Command-line compiler or IDE with C++ support
- **Operating System:** Windows 7+, Linux, or macOS
- **Text Editor/IDE:** VS Code, Code::Blocks, Dev-C++, or any text editor

These components provide a stable foundation for implementing both the backend logic and the console interface, enabling efficient development, debugging, and testing of the Contact Management System application.

3.2.2 Hardware Specifications

The system requires the following hardware to operate efficiently:

- **Processor:** Intel Pentium or equivalent (minimum); Intel Core i3 recommended
- **Memory (RAM):** 512 MB minimum; 1 GB recommended
- **Storage:** 50 MB minimum for executable and project files
- **Display:** Any standard console resolution (800×600 or higher)
- **Input Devices:** Standard keyboard for text entry

These minimal requirements ensure smooth execution of the system on almost any computer, providing a responsive and user-friendly experience while interacting with the menu and performing contact management operations. Since the

application is console-based with no graphics or animations, it runs efficiently even on older hardware without any performance issues.

Chapter 4 Discussion on the achievements

4.1 Project Overview

The Contact Management System project is a C++ console application that integrates three fundamental data structures into a single functional program. The application features a clean, menu-driven interface with numbered options for all operations. Users can interact with the Linked List, Stack, and Queue structures through intuitive menu choices for adding contacts, displaying contacts, searching, updating, deleting, viewing recent contacts, and showing the undo stack.

The main menu displays all available options with clear numbering from 1 to 8. The program dynamically updates all data structures based on user actions—contacts are stored in the linked list, action descriptions are pushed onto the stack, and viewed contact names are added to the queue. Each structure provides real-time feedback as contacts are added, searched, updated, or deleted, showing how multiple structures can work together seamlessly in a single application.

4.2 Implemented Features

4.2.1 DSA Implementations

Stack Implementation

The Stack is implemented as a fixed-size array (size 5) that stores the last five actions as C-strings. Each time a user adds, updates, or deletes a contact, a descriptive string like "Added Ram" or "Updated Sita" is pushed onto the stack using the push() function. When the stack is full, the oldest action is discarded by shifting all elements, making room for the new action. The pop() function retrieves the most recent action, though in this implementation the stack is primarily used for display rather than actual undo. The display() function shows actions from most recent to oldest, clearly demonstrating LIFO (Last-In-First-Out) behavior. This stack implementation shows how a simple array can be used to track history in real-world applications.

Queue Implementation

The Queue is implemented as a circular array of size 5 that stores the names of the last five contacts viewed. Whenever a user searches for a contact using searchContact() or updates a contact using updateContact(), that contact's name is added to the queue using the enqueue() function. The queue uses front and rear pointers to maintain circular behavior, overwriting the oldest name when the queue

becomes full. The displayRecent() function shows names from oldest to newest, demonstrating FIFO (First-In-First-Out) behavior. This queue implementation illustrates how circular arrays can efficiently maintain a fixed-size history of recently accessed items.

.

Linked List Implementation

The Linked List is implemented as a custom class with nodes containing a Contact structure (name, phone, email) and a pointer to the next node. The addContact() function creates a new node and appends it at the end of the list, demonstrating dynamic memory allocation. The displayAll() function traverses from head to tail, printing each contact's details with clear formatting. The searchByName() function performs linear traversal comparing names, showing how linked lists require sequential access. The deleteByName() function handles removal from head, middle, or end by carefully adjusting pointers. The linked list implementation demonstrates core concepts of dynamic data structures including node creation, traversal, pointer manipulation, and memory management in a practical context.

Binary Tree Implementation

The Binary Search Tree is implemented as an optional feature using TreeNode structures with left and right child pointers. Each node contains a Contact structure, and insertion follows BST rules where smaller names go to the left and larger names go to the right. The search function uses recursion to find contacts quickly without traversing the entire list, demonstrating the efficiency of hierarchical data structures. In-order traversal displays contacts in sorted alphabetical order, showing the natural ordering property of BST. While the main project uses the linked list for primary storage, this optional implementation shows how different structures offer different performance characteristics for the same data.

Sort Functionality

Each data structure includes basic organization features appropriate to its design. The linked list displays contacts in the order they were added, which is typically the order of insertion. The stack displays actions from most recent to oldest, showing LIFO order. The queue displays recent contacts from oldest to newest, showing FIFO order. The optional BST can display contacts in sorted alphabetical order using in-order traversal. These different ordering behaviors demonstrate how the same data can be organized differently depending on which structure is used and how it is traversed..

4.2.2 GUI and Interaction

Menu Navigation

The main menu displays numbered options from 1 to 8 in a clean, bordered layout. Options include Add Contact, Display All Contacts, Search Contact, Update Contact, Delete Contact, View Recent Contacts, Show Undo Stack, and Exit. The menu is displayed using simple cout statements with separator lines for visual clarity. After each operation, the screen is cleared using clearScreen() and the menu is redisplayed, creating a fresh and uncluttered interface. The pause() function waits for user input before continuing, ensuring

users have time to read output messages before the screen clears..

Control Operations

Four primary operations are available for contact management: Add Contact prompts for name, phone, and email input using cin.getline() to handle spaces properly. Display Contacts shows all contacts with formatted output including separator lines. Search Contact asks for a name and performs linear search through the linked list. Update Contact finds a contact and allows modification of phone and email. Delete Contact removes a contact by name after searching. Each operation provides clear success or failure messages, and all automatically update the stack and queue structures as needed.

Helper Functions

The clearScreen() function uses system("cls") on Windows to clear the console, providing a fresh display after each operation. The pause() function displays "Press Enter to continue..." and waits for user input, preventing output from scrolling off the screen. Input validation using cin.fail() checks for invalid menu choices and clears the input buffer to prevent infinite loops. These helper functions significantly improve the user experience despite the console-based interface, making the program feel more polished and professional.

Stack and Queue Display

The Show Undo Stack option displays the last five actions stored in the stack, showing them from most recent to oldest. Each action is displayed with a bullet point, clearly indicating the order of operations. The View Recent Contacts option displays the last five viewed contact names stored in the queue, showing them from oldest to newest. These display functions allow users to see exactly how the stack and queue maintain their respective orders, making the abstract concepts of LIFO and FIFO concrete and observable.

Header and Layout

A consistent header displays "CONTACT MANAGEMENT SYSTEM" at the top of each screen with separator lines using "=" characters. The menu is presented in a clean, numbered format with clear spacing between options. Output messages are formatted consistently with contact details shown in a structured way using labels like "Name:", "Phone:", and "Email:" with proper alignment. This clean layout ensures all information is easily readable and visually organized, making the program pleasant to use despite being console-based..

4.3 Achievements

During the development of the Contact Management System project, several key achievements were realized:

Comprehensive Data Structure Implementation

A complete set of fundamental data structures was implemented and integrated into a single application. The Linked List, Stack, and Queue classes each correctly implement their respective operations while maintaining data integrity. The linked list demonstrates dynamic memory allocation with proper node creation and deletion. The stack shows LIFO behavior by maintaining the last five actions. The queue illustrates FIFO behavior by tracking recent contacts in a circular array. The optional BST adds another layer demonstrating hierarchical organization. All structures work together seamlessly, showing how multiple data structures can collaborate in one program.

Clean Console Interface

The menu-based interface provides a clean, professional user experience. The numbered options with clear labels make navigation intuitive. The clearScreen() and pause() functions create a smooth flow that prevents screen clutter. Formatted output with proper spacing and separator lines makes contact information easy to read. Input validation prevents crashes from invalid entries. These small touches make the console application feel polished and user-friendly despite having no graphical elements..

Cross-Platform Compatibility

By using standard C++ with no platform-specific libraries except system("cls") which can be adapted, the application maintains compatibility across Windows, Linux, and macOS with minimal modifications. The code uses only standard headers like iostream and cstring, making it portable across different operating systems and compilers. The build process is straightforward, requiring only a standard C++ compiler without any external dependencies..

Educational Impact

The project successfully transforms abstract data structure concepts into tangible, observable behaviors. Users can see exactly how a linked list grows with new nodes, how a stack maintains action history in LIFO order, and how a queue tracks recent contacts in FIFO order. The optional BST demonstrates faster searching and sorted traversal. This practical demonstration helps students understand not just what each structure does, but how they can be combined in real applications, accelerating understanding and retention of these fundamental concepts.

Technical Skill Development

Development of this project strengthened proficiency in:

- C++ object-oriented programming with classes and header files
- Dynamic memory allocation using new and delete for linked list nodes
- Pointer manipulation and careful handling of next pointers
- Array-based implementation of stack and circular queue
- String handling using C-strings and strcpy/strcmp functions
- Input validation and error handling techniques
- Debugging and testing methodologies for data structures
- Code organization with separate header files and modular design

Modular Architecture

The clean separation between data structures (structures folder), user interface (ui folder), and utility functions (utils folder) demonstrates professional software design principles. Each data structure has its own header file with clear class definitions. The menu system is separated into Menu.h and Menu.cpp for better organization. Helper functions are grouped in helpers.h for reusability. This modularity makes the codebase maintainable, extensible, and easy to understand, allowing new features to be added without disrupting existing functionality.

Chapter 5 Conclusion and Recommendation

The Contact Management System project successfully accomplished its primary objective of creating an interactive console application that explicitly demonstrates three fundamental data structures—Linked List, Stack, and Queue—within a unified program, with an optional Binary Search Tree for enhanced functionality. The Linked List is implemented using custom node structures with dynamic memory allocation, clearly visualizing how nodes connect through pointers and how traversal works in a linear structure. The Stack utilizes a fixed-size array with push/pop operations, demonstrating LIFO behavior by storing the last five actions in order. The Queue implements a circular array with enqueue/display operations, showing FIFO behavior by maintaining the last five viewed contact names. The optional BST implements binary search tree insertion rules, allowing users to observe hierarchical relationships and faster searching in real-time.

The console-based menu interface provides a clean and intuitive experience, incorporating numbered options, clear screen functionality, and pause features for smooth navigation. The main menu dynamically directs users to appropriate operations based on their choices, ensuring clear feedback for all user actions. All data structure operations function correctly and reliably, with the linked list maintaining all contacts, the stack tracking action history, and the queue monitoring recent views. The result is a stable application that runs efficiently on any system with a standard C++ compiler.

5.1 Limitations

Despite its successful implementation, the project has certain limitations:

6 **Fixed-Size Stack and Queue:** The stack and queue are implemented as fixed-size arrays of five elements each. This means only the last five actions and last five viewed contacts are stored. If a user wants to see older history, it is not available. In a real application, these structures might need to be dynamic or larger.

7 **Linear Search in Linked List:** The search functionality in the linked list uses linear traversal from head to tail. This means that as the number of contacts grows, search time increases proportionally. With hundreds or thousands of contacts, searching could become noticeably slow without the optional BST enabled.

8 **No Data Persistence:** The application does not save contacts to a file between sessions. When the program closes, all contacts, action history, and recent views are lost. This limits the practical usability of the system as a real contact manager.

9 **Simple Console Interface:** While functional, the console-based interface lacks visual appeal. There are no colors, graphics, or advanced UI elements. Some users might find the text-only interface less engaging compared to modern graphical applications.

10 **Limited Undo Functionality:** The stack only stores action descriptions, not the actual data needed to revert changes. A true undo feature would require storing the previous state of contacts before each modification, which is not implemented in the current version.

11 **No Input Validation for Phone/Email:** While the program handles menu input validation, it does not validate phone number formats or email address formats. Users could enter invalid data without any warning or correction.

11.1 Future Enhancement

The current implementation of the Contact Management System provides a stable and educational platform, but several enhancements can further improve functionality, interactivity, and learning value:

12 **File I/O for Data Persistence:** Add file handling to save contacts to a text or binary file when the program exits and load them automatically on startup. This would make the application a usable real-world contact manager rather than just a demonstration.

13 **Dynamic Stack and Queue:** Replace the fixed-size arrays with dynamic implementations using linked lists so that the stack and queue can grow as needed without losing older data. This would provide unlimited action history and recent contacts.

14 **True Undo/Redo Functionality:** Enhance the stack to store complete contact states before modifications, allowing users to actually undo changes by reverting to previous versions. Add a redo feature using a second stack for better functionality.

15 **Graphical User Interface:** Upgrade the console interface to a graphical UI using libraries like raylib, Qt, or SFML. This would allow visual representation of the linked list with boxes and arrows, making the data structures even easier to understand.

16 **Advanced Search Algorithms:** Implement the Binary Search Tree properly or add a hash table for instant contact lookup by name. This would demonstrate the performance benefits of different data structures for search operations.

17 **Sorting Functionality:** Add the ability to sort contacts by name or phone number using sorting algorithms like bubble sort, insertion sort, or merge sort. This would demonstrate algorithm concepts alongside data structures.

18 **Input Validation:** Add validation for phone numbers (digits only, proper length) and email addresses (must contain @ and .). This would make the application more robust and user-friendly.

19 **Multiple Contact Books:** Allow users to create multiple contact lists or categories, demonstrating how data structures can be organized hierarchically or in collections.

References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
2. Sahni, S. (2005). *Data Structures, Algorithms, and Applications in C++* (2nd ed.). Universities Press.
3. Stroustrup, B. (2013). *The C++ Programming Language* (4th ed.). Addison-Wesley.
4. GeeksforGeeks. (2026). *Linked List Data Structure*. Retrieved from https://www.geeksforgeeks.org/data-structures/linked-list/
5. GeeksforGeeks. (2026). *Stack Data Structure*. Retrieved from https://www.geeksforgeeks.org/stack-data-structure/
6. GeeksforGeeks. (2026). *Queue Data Structure*. Retrieved from https://www.geeksforgeeks.org/queue-data-structure/
7. GeeksforGeeks. (2026). *Binary Search Tree*. Retrieved from https://www.geeksforgeeks.org/binary-search-tree-data-structure/
8. cplusplus.com. (2026). *Standard C++ Library Reference*. Retrieved from http://www.cplusplus.com/