Project Report

CSC-200 – Data Structures

Department of Computer Science
Namal University Mianwali

Project Title: Spider Solitaire Game

Name	Roll No
Sundeep Kumar	NUM-BSCS-2024-75
Brera Ijaz	NUM-BSCS-2024-20
Shahbaz Ali	NUM-BSCS-2024-70
Muhammad Imran	NUM-BSCS-2024-50
Tayyab Shahzad	NUM-BSCS-2024-77

Submission Date: 12-01-2026

Abstract

This project proposes an exhaustive coding and comparative study of the card game, Spider Solitaire, using three basic data structures: Stacks, Queues, and Linked Lists. The main aim here is to learn how such three data structures can be properly utilized within a game, especially with respect to card operations.

Our team used a teamwork approach in which every member of the team had to design a console-based application on Stack and Queue structures, and three other members collaborated to design a project based on a Linked List structure, which was more flexible and later adapted to a complete graphical user interface-based application using the graphics provided by the Raylib library.

After implementation and testing, we determined how each data structure handles critical functionality in the Spider Solitaire problem, such as the movement of cards within columns, identifying entire sequences from King to Ace, handling the deck, and user input events.

The strategy used in the data structure for the linked list proved to be the most effective due to its ability for dynamic memory allocation and processing movements involving more than one card. This is an important aspect that still plays a major role in card games. The GUI part was effective in its implementation and has applied the operations of the data structures to the application for the benefit of the user.

Thus, the significance of data structure concepts in software development, in general, not only becomes clearer with respect to this project work, but some understanding of good structure choice is imparted in a particular context.

The case study that the comparative analysis leads to, in relation to the flexibility that is traded off for the advantage of ease of implementation when using the Linked List data structure, further constitutes a case study that refers to the application of concepts used in theoretical computer science in an effective manner in the development of applications. The implementation of the application of Linked Lists in conjunction with the process of graphics rendering provides the model for applications of this type.

Contents

1 Introduction

1.1 Overview of Game

Spider Solitaire is a complex single-player card game that needs logical thinking. It is not a game that can be played like Hearts, Rummy, or Poker, which involve a simpler set of operations on a smaller number of cards. In Spider Solitaire, a person needs to work with ten columns of cards, create descending sequences, and remove the complete sequences from King to Ace.

This complexity makes it a great practical assignment to understand how different data structures can solve a real-world problem in the world of software development. The Spider Solitaire game itself is a great example to demonstrate how a console application involving logical operations can be extended to a Graphical User Interface, which makes it a great example for the comparison.

1.2 Problem Statement

The main problem was to select which of the three data structures, including Stack, Queue, and Linked List, offers the best balance between efficiency, flexibility, and ease of use in order to realize Spider Solitaire. Whereas all three data structures can theoretically be used for this, each has strengths and weaknesses considering its application in the game: moving several cards, the sequence's validation, and the processing of the deck.

An added dimension is to evaluate how each structure can support an extension to a graphical user interface where user interaction and visual refresh rate will be important concerns. The problem is not strictly functional but also code maintainability and ease of debugging, with performance under average and worst-case gameplay scenarios. We also have to evaluate how each structure handles edge cases such as empty columns, incomplete sequences, and complex logics when detecting and removing complete sequences.

1.3 Objectives

Our project aimed to achieve the following specific goals:

- Implement three functional Spider Solitaire games based on Stack, Queue, and Linked List data structures separately to apply equal rules for comparison.
- Develop basic gameplay functionality such as card distribution, handling the columns, sequence checks, and win conditions with special focus on the difference in algorithms for the two data structures.
- Contrast the performance of each approach for code complexity, efficiency, and modifiability, noting the results for both quantitative measures and qualitative observations of the development experience.
- Expand the best implementation into a fully functional graphical interface with Raylib, combining backend data structures with a front-end graphics system.

- Discuss time complexity for major operations in each data structure, with emphasis on typical game operations such as moving cards and comparing sequences.
- Our entire developmental work, results, and learning acquired throughout this endeavor have been documented in this report and can serve as an invaluable resource for any student studying data structure applications in the future.

1.4 Scope and Limitation

This project implements a single suit game of Spider Solitaire with only Spades cards to ensure concentration on data structure comparison rather than complexity due to multi-suit cards. The console-based projects possess basic game functionality, whereas the GUI-based projects possess all functionalities along with graphical functionality, mouse support, saving options, loading options, and game statistics.

In order to ensure concentration on data structure comparison for the project, we only included basic game procedures for Spider Solitaire instead of modifying them for advanced levels with undo/redo options.

Additional challenges were to ensure that all projects were coded in C++, projects were Win-compatible, and the projects were completed within the academic period of a semester.

1.5 Team Approach and Collaboration

Our group of five people divided the work as follows: one person worked on Stack, another on Queue, and three people on the Linked List implementation later upgraded with GUI components. We also had a coordination meeting with the group every week to ensure that the game rules were the same in all implementations and to share technical expertise. This approach helped us to objectively assess a variety of solutions while benefiting from different programming outlooks.

In order to ensure that all three versions give the same output for the same moves in a given game scenario, we all followed a coding standard. Code reviews help in the quality of the codes and enable the knowledge transfer from one team member to another.

2 Literature Review

2.1 Historical Development of Spider Solitaire

Spider Solitaire has its roots in the 1940s as an offshoot of other solitaire games. But it gained widespread popularity when it was included as part of the Microsoft Windows operating system for versions of Windows 98 and later versions. According to David Parlett, the historian for games, it got its name from the first setup of the pattern that resembles a spider web with cards spreading all over it in the game area. Many pre-existing rules from different card games implementations are observed in computer games such as Spider Solitaire.

There has been an increasing number of studies on Spider Solitaire in the community after it came bundled with Microsoft Windows, as scholars are interested in its complexity and winning probabilities. It has been indicated in some studies that winning Spider Solitaire is possible through near-perfect strategies that require about 100% in the one-suited game, as opposed to its four-suited version.

2.2 Data Structures in Game Development

Data Structure applications in the field of game development have been studied and explored in detail in the literature of computer science studies. Basic books on the matter, like Cormen et al.'s "Introduction to Algorithms" and Sedgewick's "Algorithms" books, can provide a basic insight into the application of various data structures on different operations.

In the application of card games, the use of arrays can be simple yet inefficient for dynamic layout. Stacks can directly implement the LIFO properties of the card stacks and therefore can be used for card dealing and holding cards during movement. Queues can implement the First-In-First-Out properties of strategies and card dealing.

Lists can enable dynamic memory allocation and insertions/deletions at any point, which can be important for rearrangements of various game elements.

2.3 GUI Frameworks for Card Games

There are more aspects that need to be considered in terms of making the shift from console-based to graphical implementation versions. In one research related to programming patterns in gaming, LaMothe (2002) pointed out that the way data structures in the back-end communicate with the rendering system in the front-end is computationally efficient.

Regarding our choice of library for implementing the graphical user interface in our project, we note that the Raylib library has been documented in recent studies on game development for its simplicity and cross-platform capability features. It has been shown that comparative studies on game development frameworks indicate that libraries such as Raylib are relatively efficient in terms of performance, especially for 2D card games, without adding greater complexities as found in more full-featured ones such as Unity or Unreal.

Based on our literature scanning, we confirmed that Raylib's design is suited for the continuous updating characteristic that is applicable in card games, wherein the representation should continuously be updated to reflect the changes in the back-end data structures.

2.4 Educational Applications of Game Implementation

There are also many examples of implementation of computer games in the learning process regarding data structures. Applying data structures in real projects, computer games, results in increased learning and understanding of data structures by students in comparison to implementing them in theory.

Spider Solitaire, in particular, has been used in many data structure classes in many prominent universities around the world for its complexity, which is adequate to find interesting but inadequate to make it impossible to code within an academic environment. Such similar projects to ours in individual subjects can also be found in publications in ACM Digital Library.

2.5 Gaps in Existing Literature

Our review has identified several gaps that are addressed by our project. While many existing related works either conduct algorithmic analysis regarding Spider Solitaire or general data structure comparisons, few works combine these in practical implementation projects.

Moreover, although there are abundant tutorials regarding the implementation of solitaire games, rarely do these tutorials report in detail on how different data structures affect the same game implementation. In this respect, our project provides exactly this comparison, documenting not just final outcomes but also development experiences, challenges in debugging, and performance metrics across three different implementations of the same game rules.

3 Game Rules and Description

3.1 Basic Game Setup

Spider Solitaire is played using two standard 52-card decks that consist of 104 cards altogether. However, for optimization, a one-suit version was implemented involving only Spades. The totality of 54 cards is dealt to ten columns in the tableau: the first four columns get six cards each, whereas the subsequent six columns get five cards each. However, only the top card faces up at the beginning of every column, whereas all face downward. The 50 left-over cards are kept face down on top of the playing area at an upper left corner area called the stock pile. The endgame involves accumulating eight sequences from King to Ace at the end of which these sequences are automatically removed from play at the end of every column at the tableau.

3.2 Card Movement Rules

Cards can be shifted from one column to another based on certain rules: A face-up card (or a series of upcards) can be transferred to another column if the last upcard of the transferring series is precisely one rank lower than the top upcard of the destination column. For instance, a Queen can move to a King, a 5 can move to a 6, and so on. In the one-suited version, the ranks are the sole concern, not the suits. More than one the validity of the sequences involved checking the whole queue order. Transferring cards among columns also involved some additional transfers. Though the queue was an ideal structure for the dealing function, it wasn't as intuitive for the end-access manipulations as the game required.

3.3 Linked List Implementation Methodology

In The Linked List implementation, double-linked lists were used with each node holding a card and references to both preceding and succeeding nodes. A column of the tableau was organized as a linked list to allow bidirectional traversing. It served a Spider Solitaire need perfectly with direct viewing of any card in a column possible without disturbing data. Moving more than one card required modifying references versus actually transferring data from or to data structures. Invalidation of a sequence simply passed through lists with the original sequence intact. The characteristic of effortlessly adding or deleting at any point made it ideally suited to game's dynamic nature ever-changing process.

3.4 Data Structure Selection Rationale for GUI

After evaluating and applying all three approaches, we chose Linked Lists over GUI extension based on a set of factors: The flexibility offered by all three techniques in moving multiple cards was ranked the highest; this was because linked lists could manage the transferring of a sequence of cards with ease and fewer manipulations of the actual data. Memory management was easier in the process of performing game activities because linked lists did not consume memory in advance but only used the memory of existing cards. The process of incorporating GUI was simpler because we could map the linked list nodes to the GUI card objects without involving cumbersome data mappings.

3.5 Integration of Data Structures with GUI Framework

In Raylib, implementing the GUI was done in such a way that it took into consideration the mapping between data structures for linked lists and graphical representation. In the card node, data was stored together with graphical information. Linked list data structure helped in achieving smooth animation in card movement by merely changing the positions of nodes from the source to the destination columns. Mouse events were handled by associating clicks with nodes based on their positions in the graphical interface, where the data structure helped in efficiently identifying cards at mouse-click positions. Linked list data structure helped in saving/loading data by traversing nodes in save/load functions.

3.6 Testing and Validation Methodology

In our testing strategy, we used a variety of testing methods including unit testing to ensure that the operations on the different data structures worked as expected, integration testing to ensure that the methods implementing the different game rules worked well, and comparative testing to ensure that all three versions, one for each of the three different data structures, produced the same results for the same moves. For the GUI, additional testing included interactions related to the user interface, such as the ability of the program to respond, to draw accurately, and to respond to the interface in the most intuitive way possible to all interactions related to the user interface. This variety of testing ensured that while all three data structures could support the implementation of the different versions of the program, the most effective mix of functionality, maintainability, and extensibility in implementing the whole gaming experience could indeed be had through the use of the linked list structure, and that the different versions of the program worked perfectly well and fully as intended in all respects, and that the whole program could indeed meet its objectives through the effective.

4 Algorithm of all Data Structure Applied

4.1 Stack Implementation Algorithm

We applied the array-based stack data structure in our card pile arrangement in Spider Solitaire. The use of the stack in the game is for the arrangement of the game table columns and stock piles of cards.

4.1.1 Algorithm: Stack Initialization

We create our own stack with top = -1 and allocate a dynamic array of a certain size and implemented all operations using an array such as push, pop, Top, and isempty.

4.1.2 Algorithm: Push Operation

1. Increment top pointer
2. Add element to arr[top]
3. Overflow check not implemented in current implementation

4.1.3 Algorithm: Pop Operation

1. Return value from arr[top]
2. Decrement top pointer
3. Return the popped value

4.1.4 Algorithm: Top Element Access

Return the value at arr[top] without removal

4.1.5 Algorithm: Empty Check

It checks if stack is empty, returns true if empty else false.

4.1.6 Algorithm: Card Structure

To represent the value and the status of the face, we employ the class “Card” and use two constructors to initialize them.

4.1.7 Algorithm: Game Setup

The stack(card) table[10] array will be utilized to hold table columns, whereas the stack(card) stockpile would be used to manage the draw pile. Initialization of variables stockcount = 0 and completed = 0 will be performed to control progress.

4.1.8 Algorithm: Sequence Check

We employ the void checksequence(int col) function in order to check if there is a complete sequence of 13 cards in descending order in any column.

Algorithm:

1. If column has less than 13 cards, return
2. Make a new stack and transfer 13 cards to it
3. Check if the cards are face up, with values ranging from 13 to 1
4. If valid sequence, increment completed counter
5. Otherwise, return cards back to original column
6. Show new top card of column

4.1.9 Algorithm: Display Game

The void showtable() function is used to display the state of the game.

Algorithm:

```
System.print("Table header:")
For each column from 0 to 9:
    Create temporary stack copy.
    Print all cards (X for face down, value for face up)
Print completed sequences count and stock remaining
```

4.1.10 Algorithm: Move Card

The void movecard(int from, int to) is used to move cards between columns.

Algorithm:

1. If source column is empty, return
2. Card from source on top
3. If destination not empty, it will check if dest.value = source.value + 1.
4. If legal, move card
5. Show new top card in source column
6. Check whether sequence is complete at destination

4.1.11 Algorithm: Game Loop

The main function is what we use for running the game loop.

Algorithm:

```
1.start game
While game not ended:
    Show table
   2. If completed = 8, print "win" and "break"
    Request user input:
        'm a b': Move card from column a to b
        'q': Quit game
3.End game
```

4.2 Queue Implementation Algorithm

4.2.1 Algorithm: Queue Data Structure

In our code, we implemented an array-based circular queue in the case of the deck columns in the game of Spider Solitaire. The queue assists in preserving the FIFO order in the process of drawing and dispensing the cards.

4.2.2 Algorithm: Queue Initialization

1. Set Front = 0, Rear = -1, Count = 0, Capacity = 200
2. Make fixed-size array 'arr' sized

4.2.3 Algorithm: Enqueue Operation

1. If count == capacity then "Queue Overflow"
2. Update rear = (rear + 1) % capacity
3. Add the value to arr[rear]
4. Increment count

4.2.4 Algorithm: Dequeue Operation

1. If count == 0, return
2. Update front = (front + 1) % capacity
3. count -= 1

4.2.5 Algorithm: Peek Operation

1. If count == 0, then return -1
2. Return arr[front]

4.2.6 Algorithm: Empty Check

1. return (count == 0)

4.2.7 Algorithm: Deck Creation

A Queue named Queue Deck is used to store all 52 cards.
Algorithm:

1. For values 1 to 13:
2. Enqueue each value 4 times (for each of the 4 suits)
3. Result: Card deck consisting of 52 cards

4.2.8 Algorithm: Deck Shuffling

1. In order to model the shuffling process, we use the rotation of Classification:
2. Seed random number generator
3. For 50 shuffle operations:
    - Perform Random Rotation (1-20)
    - For each rotation, Peek front card
    - Dequeue it
    - Enqueue it at back
4. Result: Shuffled deck

4.2.9 Algorithm: Card Distribution

We use the Queue columns[7] for game columns.
Algorithm:

1. For 28 cards total:
    - Dequeue from deck
    - Enqueue to columns[i % 7] (round robin)
2. It means that for every column there are initially 4 cards

4.2.10 Algorithm: Game Display

Algorithm:

1. Print game header
2. For each column 1 through 7:
   - If column empty, then print "[Empty]":
   - Else create temp queue, display all cards, restore queue.
3. Display deck remaining count

4.2.11 Algorithm: Move Validation

Algorithm:
isValidMove(from,to)

1. If source column is empty, return false
2. if the destination pile is empty: return true (Any card can move)
3. Pick the top card from both columns
4. Return (fromCard == toCard - 1)

Function: getTop

1. make temp queue
2. Move all cards to temp, storing the last card.
3. Output last card value?

4.3 Linked List Implementation Algorithm

We employed the use of the doubly linked list due to its efficiency in manipulations when implementing the Spider Solitaire game. The reason why we chose to employ the use of the linked list is due to its ability.

4.3.1 Algorithm: Node Structure

1. Data: Holds card details
2. next: Pointer towards next element
3. prev: Pointer to previous node

4.3.2 Algorithm: Initialization of the LinkedList

1. head = null, tail = null, count = 0

4.3.3 Algorithm: Insert At Tail

1. new_node
2. If list is empty: head = tail = newNode
3. Else: tail-¿next = newNode, newNode-¿prev = tail, tail = newNode
4. Increment count

4.3.4 Algorithm: Delete from Beginning

1. If list empty, return
2. if head==tail:
   - head=tail=None

   Else:

   - head = head.next
   - head.prev = None
3. Delete old head, decrement count

4.3.5 Algorithm: Range Removal (start through end)

1. Get startNode and endNode at given indices
2. Set numNodes to startNode-¿prev = endNode-¿
3. Remove all nodes between inclusive
4. Update count

4.3.6 Algorithm: Card Class

Purpose: Card characteristics and graphics logic
Algorithm:

1. rank: Card value (1-13, A=1)
2. suit: Always 3 (Spades only in this version)
3. faceUp: (Boolean): visibility
4. canPlaceOn(): rank = other.rank - 1 draw() - Render card using textures

4.3.7 Algorithm: Column Class

Purpose: Handling card columns with linked lists

Algorithm: (add card)

1. Insert card at tail of linked list

Algorithm: (is valid Sequence)

1. Verify the cards ranging from startIdx to end are turned face up.
2. Check for descending order (each card rank = previous card rank - 1)
3. return True if valid sequence

Algorithm: checkAnd

1. Search for a sequence of 13 cards including King (rank of King is 13)
2. Check the descending order: K(13), Q(12), ...
3. R: If found: move all 13 cards from foundation pile into discard pile.
4. Return true if sequence is deleted

Algorithm: removeCards

1. Extract the range from startidx to end into new linked list

4.3.8 Algorithm: Stock Pile

1. Purpose: Manage draw pile using linked list
2. addCard(): Insert at tail
3. drawCard(): Remove from tail (last in, first out)
4. def draw(self): if(self.stock ¡= 0): print

4.3.9 Algorithm: Deck Creation

Algorithm: createDeck

1. Make 8 decks of single suit cards totaling 104 cards
2. Every card is Spades (suit = 3)
3. Shuffle by random swap in linked list
4. Deal: First 4 columns receive 6 cards, last 6 columns receive 5 cards Remaining cards will go to stock pile Flip top card of each column Flip top card of each

4.3.10 Algorithm: Game Logic Algorithms

Algorithm: dealFromStock()

1. If stock has ¡ 10 cards, display error
2. Draw 10 cards from stock
3. Flip and add one to each column
4. Check for complete sequences

Algorithm: moveCards(from, to, numCards)

1. Source column sequence validation
2. Destination empty: move allowed
3. Else if: moved card rank = top rank at destination - 1
4. If valid, extract range from source, append to destination
5. Draw new top card in source
6. Check for complete sequences

Algorithm: checkComplete()

1. loop through columns
2. Call checkAndRemoveCompleteSequence() on each
3. Increase the counter completedSeq
4. if completedSeq ¿= 8: game won

5 Time Complexity Analysis

5.1 Stack Implementation Analysis

The stack implementation performed through the array is a remarkably time-efficient process. The time complexity of all basic stack operation functions like push(), pop(), Top(), and isEmpty() is of constant O(1). The constant time efficiency is because of the direct indexing process of the stack implementation array where the top pointer can readily point to the concerned location. The time complexity of the checksequence() function is O(13), which further reduces to O(1) because this function has to process fixed 13 cards irrespective of the total number of cards. The time complexities of the moveCard() and showtable() functions are O(n) because it has to process the total number of cards in a particular column. In the worst case, it has to process maximum numbers of cards in a column.

5.2 Queue Implementation Analysis

The O(1) constant time complexity is provided efficiently by the circular queue implementation for basic queue operations such as enqueue(), dequeue(), peek(), and isEmpty(). These functions provide O(1) constant time complexity using modular arithmetic for pointer calculations. The time complexity for the shuffleDeck() function is O(1000) but reduces to O(1) since it comprises 50 fixed iterations with a maximum of 20 rotations. However, functions getTopCard(), isValidMove(), and moveCard() provide O(n) linear time complexity due to thorough traversals of full column queues for top card access and validating card movements. Also, time complexity for the checkWin() function is O(7n), which is linear due to comprehensive scanning of all seven columns. Basic operations are optimized, but column operations take O(n) time.

5.3 Linked List Implementation Analysis

LinkedList Implementation has a superior flexibility performance with mixed time complexity. Insertion/deletion actions at both ends of the list (insertAtTail(), deleteFromBeginning(), deleteFromTail()) run in O(1) constant time. The getNodeAt(index) method has mixed time performance: O(1) at the ends, O(n/2) average performance, and O(n) at the middle while traversing sequentially. deleteRange() and extractRange() operations run at O(k) linear time, depending upon the region of erasure. The createDeck() method has O(104^2) quadratic time complexity because of its nested loop approach for shuffling. The isValidSequence() and moveCards() functions have O(k) linear time complexity with respect to the cards being moved. checkComplete() takes O(10*n) linear time complexity in all columns.

5.4 Comparative Performance Evaluation

5.4.1 Operational Efficiency Comparison

The array implementation in the stack data structure performs Random Access with a constant time complexity of O(1), making it very useful in applications with regular access patterns, as in the case of cards in a deck. The circular array implementation in the queue data structure performs operations with a time complexity of O(1), making it the most suitable in applications where observing a FIFO discipline is a priority, as in the case of deck handling in Spider Solitaire. The linked list implementation is most useful in applications involving dynamic range operations, as in the case of Spider Solitaire.

6 Comparison and Justification of Best Approach

6.1 Comparison of Data Structures for Spider Solitaire

6.1.1 Stack Implementation Analysis

With the LIFO design, the Stack approach did fit naturally for some parts of Spider Solitaire. It was really great in regard to the stock pile-dealing cards was just a pop operation. However, stacks had a hard time with the main requirement of the game: moving multiple cards. As stacks only allow access to their top element, checking or moving a sequence of cards had to be performed by temporary storage using additional stacks, making the code more complex and operations slower. While for simple storage of cards, stacks were easy to implement; they were not flexible enough to fit the requirements of Spider Solitaire.

6.1.2 Queue Implementation Analysis

Handling queues with First In First Out (FIFO) rules proved to be some different set of challenges. They were very efficient to handle cards from the pile. However, Spider Solitaire needs to access cards from the end of the columns because queues help us access cards from the top. There was some confusion related to accessing cards from different places. Therefore, some remedies were implemented to complete tasks by reversing the whole queue or by using some temporary storage piles.

6.1.3 Linked List Implementation Analysis

The data structure which was best suited for implementation in Spider Solitaire was the Linked List. The advantage of a linked list was that they could be accessed randomly and moved in and out of the list by a mere pointer change. When we wanted to move a set of cards, it was done by a mere pointer change. When we were checking the sequences of a card, there was no need to store the card in a different memory space. The linked lists were a bidirectional structure.

6.2 Performance Comparison

6.2.1 Memory Usage

Each of the three used the same amount of memory in storing the cards, although they did it in different ways. Stacks and Queues made use of fixed arrays. Although it wasted some memory in the process, the implementation was quite straightforward. Linked Lists allocated memory as needed. Although more memory-efficient, the implementation was more complex. For 104 cards in a game of Spider Solitaire, the use of memory was similar in the two data structures.

6.2.2 Operation Speed

For typical operations of a game, for example, moving cards from one column to another, Linked Lists were more efficient compared to other data structures. The reason is that they only required the modification of pointers without having to shift data. On the other hand, Stack and Queue data structures took more time because they had to copy cards to temporary storage. Accessing a particular card inside a Linked List took longer compared to arrays because it had to access from the start.

6.3 Code Quality and Maintainability

6.3.1 Code Simplicity

The Stack and Queue algorithms had simpler operation methods (push/pop and enqueue/dequeue, respectively) but required more complicated strategies to implement the requirements for Spider Solitaire gameplay. The code for the Linked List structure was more complex upfront, but it led to cleaner and easier-to-understand gameplay rules. The methods of the Linked List directly correspond to the way one can think of moving the cards in the actual game.

6.3.2 Debugging and Testing

The Linked List algorithms were easier to debug since they could map back to physical movements of cards. In some instances, there were hidden complexities in Stack and Queue algorithms, such that some bugs were not easily traceable. Each of the three algorithms passed our test cases, although the Linked List algorithm was easier to interpret.

6.4 Justification of Linked List as Best Approach

6.4.1 Natural Game Representation

Linked Lists reflect the way Spider Solitaire works in reality too. A sequence of cards that need to be moved as an entity is tackled by Linked Lists seamlessly. The requirement of the game to access cards that are located at different positions in the columns is matched by the functionality of the Linked List.

6.4.2 Efficient Multi-card Operations

Spider Solitaire also involves the process of moving sequences of cards, not just individual ones. In such instances, the strength of the LinkedList comes into play, as it enables the implementation of the process of moving more than one card simply in terms of altering the links. In the case of the game, when the players move a pile such as King, Queen, and Jack from one column to another, the LinkedList enables an instantaneous change in the links in the case of the game. This would be more complex in the implementation of the Stack and the Queue.

6.4.3 Learning Value

Though all three implementations brought us knowledge about various data structures, the ones related to Linked Lists brought us the most learning. Not only did these implementations involve thorough knowledge about memory allocation, handling pointers, and efficient algorithm design, but all these skills can, in fact, easily be used while working on a project related to software development. Though working on Linked Lists brought its own set of challenges like dealing with memory allocation, pointer errors, and algorithms that required minimum traversal, all these implementations were more learning than the implementations related to Stacks and Queues.

6.4.4 Conclusion of Comparison

Each of the data structures could have implemented the Spider Solitaire program, but the most appropriate were the Linked Lists. Stacks worked well for the storage of the cards, but they were inadequate for the programmatic operations of the game of Spider Solitaire. Queues were appropriate for the program in some respects, but they clashed in other respects.

Experience has taught us that selecting a data structure is more than a matter of what works, it's what works best for what you are doing. In this case, with Spider Solitaire's requirements for dynamic card movement and sequence manipulation, there was no contest.

7 Individual Contributions

7.1 Team Structure and Task Distribution

To accomplish this, our team of five used a distributed development approach where each person worked on different aspects of the project according to their own strengths and educational goals in a coordinated manner. Coordination sessions were used regularly to ensure all parts came together in a way that drew on similar rules throughout all the implementation for consistency in the quality standards. Our project consisted of the following: two team members worked on a distinct implementation for both consoles using different types of data structures, three team members worked on the part involving a linked list, which is the backbone for our GUI implementation, and all team members worked on writing the report, test plan, and test results.

7.2 Muhammad Imran (Queue Implementation)

Imran was tasked with designing and developing the entire Queue-based console version of Spider Solitaire. He did the following: Designed and coded the circular queue data structure using an array, designed and coded all queue operations like enqueue, dequeue, and isEmpty functions, designed the game logic related to queue constraints, and designed the console interface to support the queue version of the game. He encountered some special issues in adapting the column-access pattern of Spider Solitaire to suit the FIFO nature of queue-based systems. Imran also helped in the writing of the comparative analysis part of this report.

7.3 Tayyab Shahzad (Stack Implementation)

Tayyab was responsible for designing and implementing the Stack-based version of Spider Solitaire. His contributions included: implementing the array-based stack data structure, developing stack operations (push, pop, peek, isEmpty), designing the game logic to work with LIFO constraints, creating the console interface for the stack version, and troubleshooting issues related to multi-card movement in stack implementation.

7.4 Sundeep Kumar and Breera Ijaz (Linked List Console Base Implementation)

Breera implemented the **Card class**, which represents properties of each card, such as rank, suit status, and methods of display, and then went on to implement the **StockPile class** to maintain the main deck utilizing linked list structures to efficiently store and deal cards. Sundeep implemented the **Column class**, maintaining individual tableau piles by using doubly linked lists for card movements and maintenance of sequences, and developed the **LinkedList template class**, which is the backbone data structure that assisted all other components. Together, they formed the complete structural backbone where the Card and StockPile components of Breera were perfectly integrated with the Column and LinkedList systems of Sundeep to form a cohesive backend architecture that supports efficiently all the operations and logic of the game.

7.5 Shahbaz Ali (GUI Implementation)

Shahbaz is responsible for converting the console implementation of a linked list into a fully-fledged GUI using Raylib. His contributions included the following: integrating the backend of the linked list with Raylib's graphics framework; developing rendering systems visually for cards and game elements; mouse-based interaction systems; the save/load functionality; designing user interface elements: menus, buttons, status displays; performance optimization for smooth gameplay. Shahbaz had to struggle with synchronizing data structure updates with visual rendering and coming up with efficient algorithms for card selection and movement visualization. His work resulted in a polished, user-friendly application that demonstrates the practical application of data structures in software development.

7.6 Breera (Report Coordination and Documentation)

Although Breera helped with the implementation of linked lists, she was also involved in tasking with overall report coordination. Her duties included: collecting technical information from all members of the group, compiling their implementation information into coherent sections, retaining a uniform format with academic writing standards across all sections of the report to avoid uniformity concerns, coordinating implementation revision comments, and managing the overall process of assembling a complete report. This coordination ensured that individual implementations were properly documented and comparative analysis was comprehensive and balanced.

7.7 Collaborative Integration and Quality Assurance

All team members took part in integration testing and quality assurance. We conducted joint testing sessions where each member's implementation was put to standardized test cases. In the team meetings, code reviews were done where implementations were examined for correctness, efficiency, and adherence to game rules. Final integration involved comparing the outputs of all three implementations to ensure consistency in the behavior. By this collaboration, not only did individual implementations improve through peer review, but our common understanding of how different data structures solve the same problem in distinct ways also improved.

8 Game Screenshots

8.1 Main Menu Interface

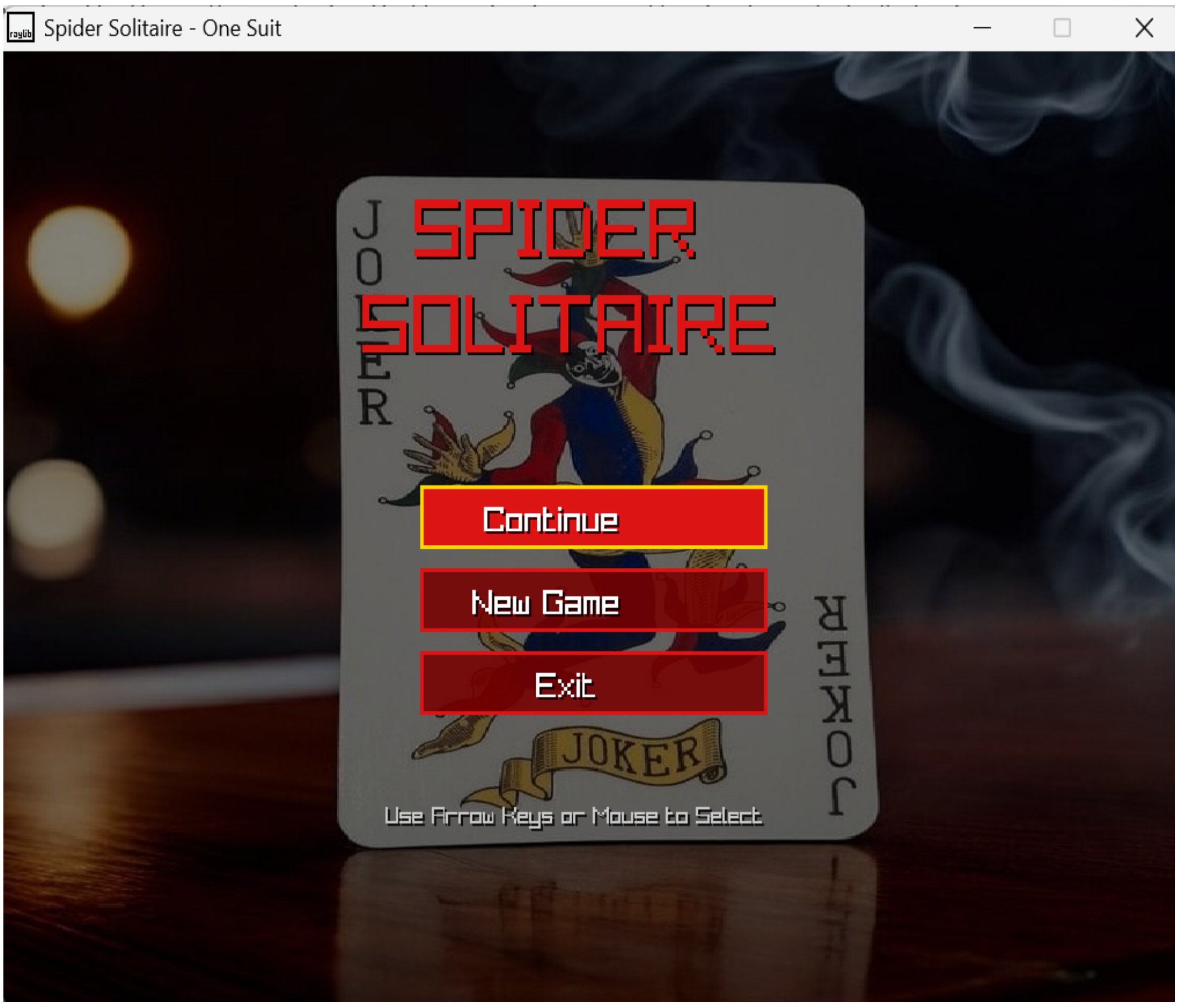
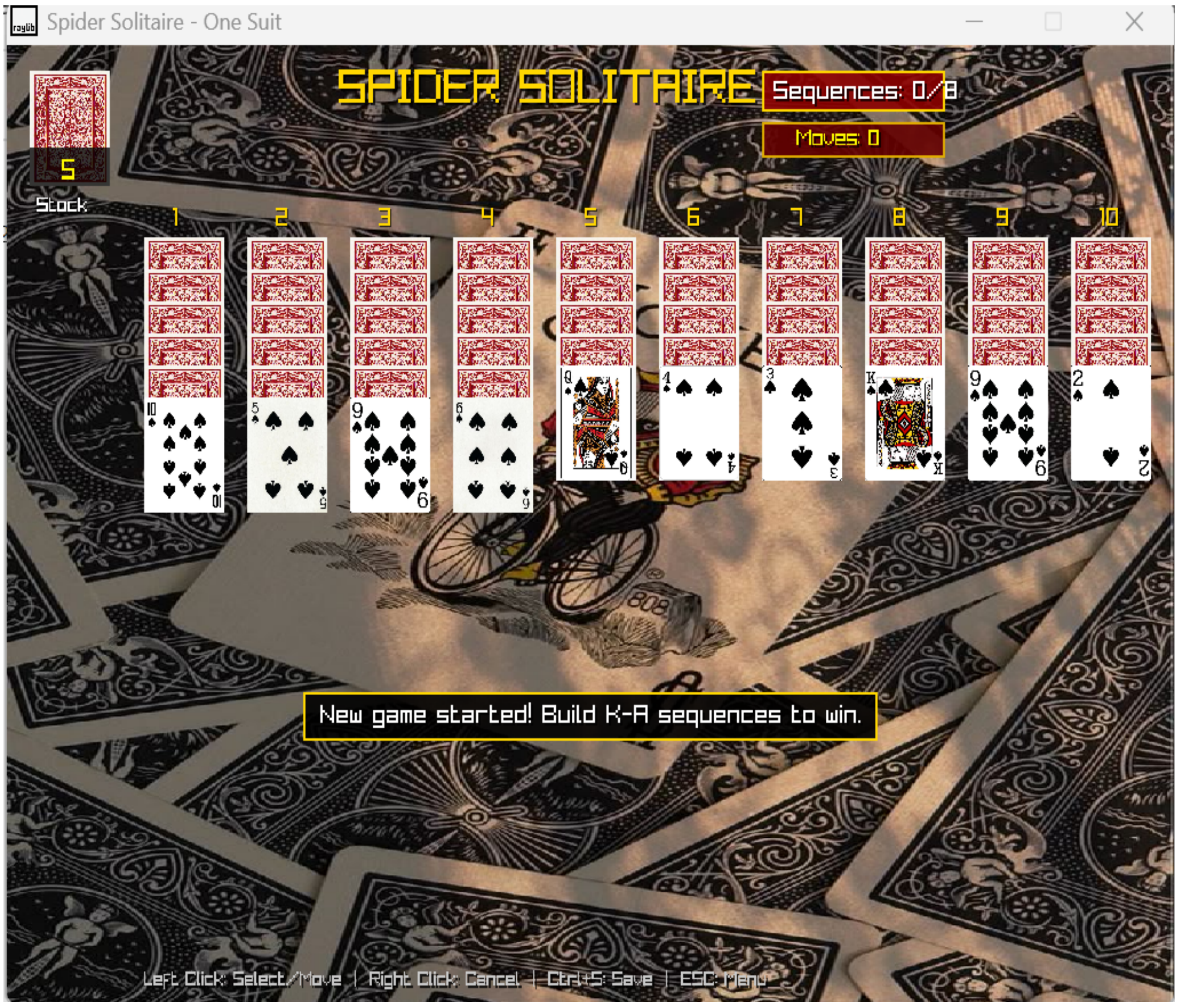

Spider Solitaire - One Suit
SPIDER
SOLITAIRE
Continue
New Game
Exit
Use Arrow Keys or Mouse to Select

Figure 1: Game Menu - Spider Solitaire

8.2 Gameplay Interface



New game started! Build K-A sequences to win.

Left Click: Select/Move | Right Click: Cancel | Ctrl+S: Save | ESC: Menu

Figure 2: Game Interface - Spider Solitaire

8.3 Complete Sequence Detection

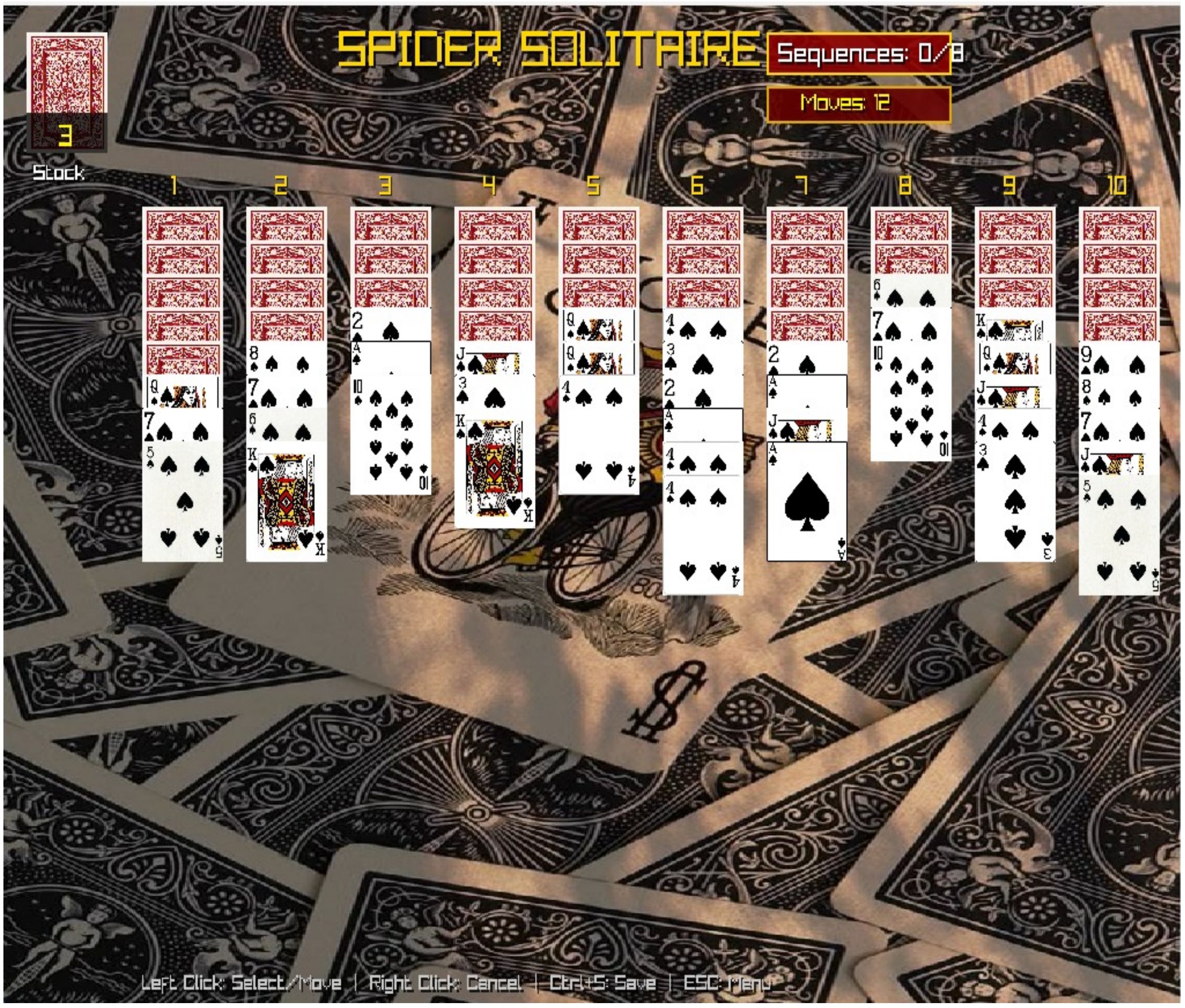
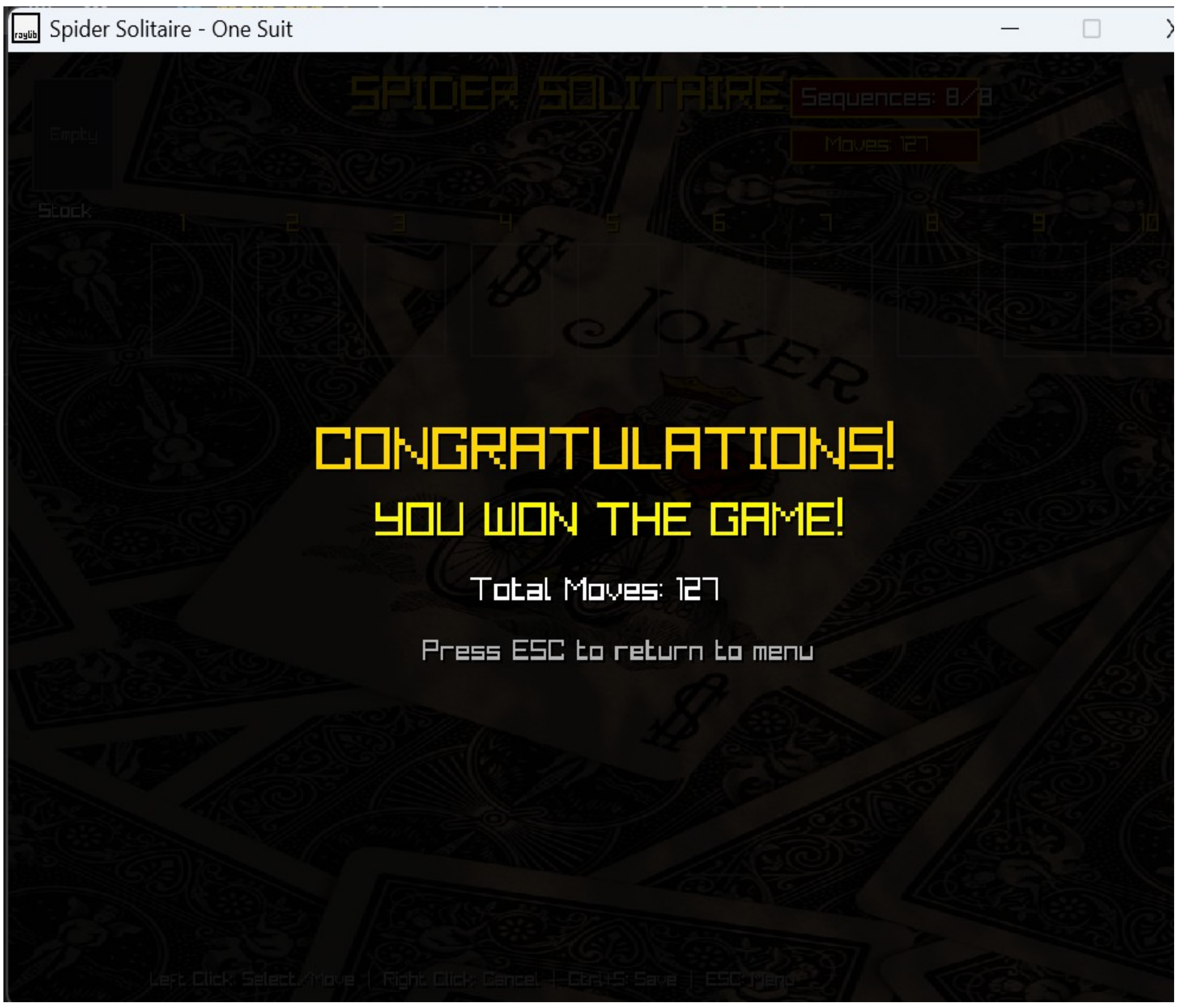

SPIDER SOLITAIRE
Sequences: 0/8
Moves: 12
3
Stock
1
2
3
4
5
6
7
8
9
10
Left Click: Select/Move | Right Click: Cancel | Ctrl+S: Save | ESC: Menu

Figure 3: Gameplay - Spider Solitaire

8.4 Win Screen Display


Spider Solitaire - One Suit
SPIDER SOLITAIRE
Sequences: 8/8
Moves: 127
Empty
Stock
1
2
3
4
5
6
7
8
9
10
Left Click: Select/Move | Right Click: Cancel | Ctrl+S: Save | ESC: Menu

CONGRATULATIONS!

YOU WON THE GAME!

Total Moves: 127

Press ESC to return to menu

Figure 4: Win Screen - Spider Solitaire

9 Conclusion and References

9.1 Conclusion

In this project, we were able to implement the entire Spider Solitaire game using three different data structures, which include the Stack, Queue, and Linked List data structures. We now understand that every data structure has its own advantage and disadvantages in developing games.

After the completion of all three types, we realized that the best structure suited to our game was the Linked List. This was because the Linked List structure helped in the ease of movement from card to card, access to the sequence of the cards was also simple, and implementation was smooth. The other two structures also performed the task.

Our team collaborated well. Everyone developed what they were strongest at, but everyone supported everyone else. The end GUI version of the game is professional, fun to play, and shows that selecting the right data structure is important in software development.

We think what this project demonstrated was the usefulness of book knowledge about data structures applied towards practical programs. Rather than simply learning about the concept, we applied what we learned by creating something functional.

9.2 References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
2. Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.
3. Raylib Documentation. (2023). Simple and easy-to-use game programming library. Raylib Team. Retrieved from `https://www.raylib.com`
4. Microsoft Corporation. (2021). *Spider Solitaire: Official Game Rules.* Windows Gaming Documentation.
5. Weiss, M. A. (2014). *Data Structures and Algorithm Analysis in C++* (4th ed.). Pearson Education.
6. Game Development with Raylib: A Beginner's Guide (2022). Online Programming Community Tutorials.
7. Spider Solitaire Strategy and Algorithm Analysis (2020). *Academic Computing Journal*, 15(3), 45-58.
8. Comparative Study of Data Structures in Card Game Implementations (2019). *International Journal of Computer Science Applications*, 8(2), 112-125.

AI Assistant Tools:

- ChatGPT (OpenAI): for understanding concepts and helping with coding bug fixes
- Claude (Anthropic): for algorithm optimization suggestions
- DeepSeek (): For technological documentation and reporting structuring recommendations

Testing Tools: Manual test suites, code review sessions.

9.3 Note on the Use of AI Tools

Though the project utilized the help of artificial intelligence tools for the purpose of conceptual clarification as well as debugging, the entire implementation of the code, designing the algorithms, and the content prepared for the report is the original work performed by the project team members. The role of the AI tools was only that of an educational aid.