NAMAL UNIVERSITY

Mianwali

Department of Computer Science
Namal University Mianwali

TriPeaks Solitaire Game Report

Implementation Using Linked Lists

CSC-200- Data Structure

Course Project Report

Submitted by:

Ahmer Sultan (NUM-BSCS-2024-03)
Najiba (NUM-BSCS-2024-61)
Najia Nayab (NUM-BSCS-2024-62)

Submitted to:

Abdul Rafay Khan

January 12, 2026

Contents

1 Introduction

1.1 What is This Project About?

This report explains how we developed a Tri-peaks Solitaire card game using linked lists as the primary data structure. In Tri-peaks Solitaire, your goal is to clear three pyramids of cards. You do this by matching cards from the peaks with a face-up waste pile, one strategic move at a time.

The development process followed a systematic approach. We began by implementing a console-based version to establish the core game logic and data structure operations. Once the fundamental mechanics were working correctly, we enhanced the project by adding a graphical user interface using the Raylib library. This two-phase approach allowed us to focus on getting the linked list implementation right before adding visual elements.

1.2 About Tri-peaks Solitaire

Tri-peaks Solitaire uses a 52-card deck.. The cards are arranged in three pyramid shapes (that's why it's called "Tri-peaks" - three peaks). The player tries to remove all the cards by matching them with a waste pile card.

2 Detailed Individual Contributions

2.1 Ahmer Sultan (03): Data Architect

Ahmer served as the backbone of the project. His primary contribution was the design of the 'Node' struct and the 'Pyramid' class.

2.1.1 Dependency Wiring

He wrote the recursive algorithm that links a card in Level 2 to its two corresponding children in Level 3.

2.1.2 Memory Management

Implemented the destructor logic that traverses all linked lists to free heap memory upon game exit.

2.1.3 Difficulty Scaling

Developed different "shuffling seeds" to allow for varying difficulty levels in the board setup.

2.2 Najiba (61): Lead GUI Developer

Najiba was responsible for the visual layer. She transformed Ahmer's console based program and functions into a playable board.

2.2.1 Texture Mapping

She implemented a graphical loading system to efficiently render 52 different card faces.

2.2.2 Scaling Logic

Designed the GUI to be responsive, ensuring that the card peaks stay centered on both 1080p and 720p displays.

2.2.3 Over-draw

Solved the "Overdraw" problem, ensuring that lower cards appear behind the top cards by sorting the draw-call list.

2.3 Najia Nayab (62): Logic Validator & Collaborative GUI Helpout

Najia's role was critical in ensuring the game followed official rules while assisting with UI feedback.

2.3.1 Match Validation Engine

She developed the core comparison function that allows an Ace to be matched with either a 2 or a King, ensuring that all related input should follow the game rules.

2.3.2 Interactive UI

Najia designed and implemented the score counter, the "Game Start" menu, and the visual indicators.

3 How the Game Works

3.1 Game Setup

The game begins with the cards arranged in this layout:

- Three pyramids of cards overlapping each other
- A stock pile (face-down cards) at the bottom
- A waste pile (face-up cards) next to the stock pile

The pyramid has four levels:

- Level 1 (top): 3 cards
- Level 2: 6 cards
- Level 3: 9 cards
- Level 4 (bottom): 10 cards

This gives us 28 cards in the pyramid. The remaining 24 cards go into the stock pile, and one card goes to start the waste pile.

3.2 Rules for Playing

The rules are simple:

1. You can only remove a card if it is not covered by any other card
2. The card you want to remove must be one number higher or one lower than the top card on the waste pile
3. If you cannot make a move, you can click the stock pile to get a new card

Example of matching:

- If the waste pile shows a 7, you can remove a 6 or an 8
- If it shows a King, you can remove a Queen or an Ace
- If it shows an Ace, you can remove a 2 or a King

3.3 Winning and Losing

You win the game by clearing all 28 cards from the three pyramid peaks.

You **lose** if:

- No cards are left in the stock pile AND
- No cards in the pyramid can match the waste pile card

4 How We Built the Game

4.1 Step 1: Console Version

We started by making a simple text-based version. This version showed the game in the command prompt window.

Why we started with console:

- Easier to test the game logic
- No need to worry about graphics first
- Could focus on making linked lists work properly

How the console version looked:

- Hidden cards showed as: **
- Revealed cards showed as: SA, H7, D9 (two letters for each card)
- Players typed commands like "SA" to remove a card or "99" to draw from stock

We tested everything in this version before adding any graphics. This made it easier to find and fix bugs.

4.2 Step 2: Adding Graphics with Raylib

After the console version worked correctly, we added graphics. We chose Raylib because:

- It is simple to use
- Works on different operating systems
- Good for making 2D games

What we changed:

- Replaced text output with actual card images
- Added mouse clicking instead of typing
- Made buttons for difficulty selection
- Added colors and shadows to make cards look real

The important thing is that we kept all the linked list code the same. We only changed how the game looks, not how it works inside.

5 Understanding Our Data Structures

5.1 What is a Node?

A node is like a container that holds information about one card. Each node contains the card name (like "SA" for Ace of Spades), the pyramid level number (1, 2, 3, or 4), a revealed flag showing if the player can see this card, a removed flag indicating if the card has been played, a next pointer to the next card in the list, and two child pointers (child1 and child2) pointing to the two cards below it in the pyramid.

Think of it like this: if you have a card in your hand, the "data" tells you what card it is, "next" points to the next card in a row, and "child1" and "child2" point to the two cards it covers in the pyramid.

5.2 What is a Linked List?

A linked list is a chain of nodes. Each node points to the next one. It's like holding hands in a line - each person holds the hand of the next person.

We used linked lists for:

- Stock pile (the face-down cards)
- Waste pile (the face-up cards)
- Each level of the pyramid (4 separate lists)

Why linked lists are good here:

- Easy to add cards to the end
- Easy to remove cards from the front
- Don't need to know the size ahead of time

5.3 How We Connected the Pyramid

The tricky part was connecting cards in the pyramid. Each card needs to know which two cards it covers.

Example:

- Card at position 0 in Level 1 covers cards at positions 0 and 1 in Level 2
- Card at position 0 in Level 2 covers cards at positions 0 and 1 in Level 3

We wrote a function called `initializeChildren()` that goes through each level and sets up these connections. It was complicated because the three peaks overlap in a specific pattern.

6 Main Functions in the Game

6.1 Shuffle Function

Before starting, we shuffle the cards. But we don't shuffle all cards - only some at the end.

How it works:

- Easy mode: shuffle last 5 cards
- Medium mode: shuffle last 10 cards
- Hard mode: shuffle last 20 cards

The shuffle uses a simple method. We go backward through the cards we want to shuffle. For each card, we pick a random position and swap them. The algorithm starts from the last card, moves backward to a stopping point, and for each card picks a random card before it and swaps their positions.

This makes the game easier or harder depending on how many cards get shuffled.

6.2 Checking if a Move is Valid

When a player clicks a card, we need to check if they can remove it.

Steps we follow:

1. Find the card the player clicked
2. Check if the card is revealed (not hidden)
3. Check if the card is not already removed
4. Compare the card with the top of the waste pile
5. See if they are one number apart

The comparison part checks all possible matches:

- Numbers 2-9 can match with one higher or lower

- 10 (shown as T) matches with 9 or Jack
- Jack matches with 10 or Queen
- Queen matches with Jack or King
- King matches with Queen or Ace
- Ace matches with King or 2

6.3 Revealing Cards

Clearing a card reveals the cards below it.

This is explained as:

- After each move, we check all cards in levels 1, 2, and 3
- For each hidden card, we look at its two children
- If both children are removed, we reveal the parent card

This happens automatically after every move. The player does not need to do anything.

Example: When you remove both Level 3 cards that are covered by a single Level 2 card, you automatically reveal that Level 2 card.

6.4 Checking for Win or Loss

Win condition is simple:

- Go through all four levels
- Check if every card is removed
- If yes, player wins

Loss condition needs more checking:

- First, see if stock pile has cards left
- If yes, player can still make moves
- If no, check every revealed pyramid card
- See if any card matches the waste pile
- If no matches exist, player loses

7 Algorithms of All Applied Data Structures

7.1 Linked Lists for Card Management

We built our game using linked lists instead of arrays. The reason is simple: we're constantly pushing cards onto stacks as the game runs. Arrays would require resizing and copying everything whenever we exceed capacity. With linked lists, we just allocate a new chunk and chain it on. Each node in the chain holds one card and points to the next node.

7.1.1 How insertAtEnd Actually Works

When we insert a card at the end, we first create a new node with memory allocation. We set the card data in that node. We look at the head pointer. If nothing is at the head, our new node becomes the head. If the head exists, we start there and follow the next pointers. We keep stepping through nodes until we reach one where next is null. That's the end. We attach our new node to that last node's next field.

This gets called constantly. Building the stock pile does this 23 times. The waste pile starts with one card. Each pyramid level array needs multiple insertions.

7.1.2 Cleanup and Memory Deallocation

When the player removes a card during play, we delete that specific node immediately. This releases heap memory right away. At shutdown, we walk all remaining lists and delete every node still there. Otherwise we leak memory. The destructor handles this cleanup phase.

7.2 Node Class Structure

Each node is not just a card container. It's more like a junction box connecting different parts of the game simultaneously.

7.2.1 What Information Each Node Stores

The data field is a string holding the card identifier. The revealed bool tells us if the card is face up or still hidden. The removed bool marks whether this card has been taken out of play. The level int records the row number. The next pointer goes horizontally to the next card in the same row. The child1 and child2 pointers go downward to cards that this card covers. One node can exist in multiple linked structures at once because of these different pointer types.

7.3 Pyramid Construction Process

Building the pyramid involves two separate stages. First we slice the deck into four rows. Then we wire up which cards sit on top of which other cards.

7.3.1 Distributing 28 Cards Across Four Rows

The stock takes the first 23 cards. Waste gets card 24. That leaves 28 for the pyramid. Array1 holds cards 24-26 (the three peak cards). Array2 gets 27-32 (six cards). Array3 gets 33-41 (nine cards). Array4 gets 42-51 (ten cards at the base). We preserve order as we insert them.

7.3.2 Creating Parent-Child Links Between Rows

Starting from the bottom: array4 has no children below it. For array3 linking to array4, we walk array3's nodes. We maintain a pointer into array4. Each array3 node's child1 points to the current array4 node. child2 points to the next one. Then we advance the array4 pointer by one. This ensures no array4 node gets missed or double-covered.

Array2 linking to array3 is different because of the pyramid shape. We need some array3 cards covered by two array2 cards, creating overlap. We loop array2 nodes. For each one, child1 and child2 point to consecutive array3 nodes. Then we skip ahead by two in array3 plus one extra spot. This gap creates the pyramid visual.

Array1 is hardcoded. The left peak covers array2 indices 0-1. The middle covers 2-3 (skipping 1). The right covers 4-5. Simple.

7.3.3 Revealing Cards After Removal

In reveal(), we iterate rows 1 through 3. For hidden cards, we test: are both children removed? If yes, flip revealed to true. Only 18 cards exist in these rows so it's instant.

7.4 Partial Shuffling by Difficulty

We don't shuffle the entire deck. That wastes time. We shuffle only the stock pile portion, and only the number of cards based on difficulty. The pyramid cards stay fixed. The waste stays fixed.

7.4.1 Difficulty Levels and Shuffle Ranges

Easy shuffles 5 cards. Medium shuffles 10. Hard shuffles 20. These are the topmost cards in the stock zone. The algorithm picks a random position in the zone and swaps. It repeats for the shuffle range size. Nothing fancy, just basic swap shuffling.

7.4.2 Why We Use Swap Shuffling

We start from the end of the shuffle zone and work backward. For each position, we pick a random card from that position down to the start. We swap them. Move to the previous position and repeat. This guarantees every card in the zone gets randomized. The time is proportional to the number of cards, which is at most 20.

7.5 Move Validation Logic

The player clicks a card. We have to check if it's a legal move against the current waste pile card.

7.5.1 Locating the Clicked Card and Extracting Ranks

We grab the last node in the waste list. This is the visible card. We search all four arrays looking for the clicked card. When found, we verify it's revealed and not removed. We extract the second character which is the rank. We do the same with the waste card. Now we have two ranks to compare.

Code 99 is a special case: drawing from stock. We remove the head of array1 and add it to array2.

7.5.2 The Matching Rules Implementation

Two ranks must satisfy certain rules. Numbers 2-9 can match if they differ by one: 3 matches 4 or 2. Ace matches 2 or King. 10 matches 9 or Jack. Jack matches 10 or Queen.

Queen matches Jack or King. King matches Queen or Ace. If nothing matches, the move is illegal. Each rank has exactly two valid partners.

7.6 Converting Mouse Position to Card Identity

Cards are drawn at specific pixel coordinates. When the mouse clicks, we need to figure out which card got hit.

7.6.1 Fixed Coordinate Tables

Array1 at x: 300, 650, 1000. Array2 at x: 230, 350, 580, 700, 930, 1050. Array3 at x: 165, 285, 405, 525, 645, 765, 885, 1005, 1125. Array4 at x: 110, 230, 350, 470, 590, 710, 830, 950, 1070, 1190. All cards are 90 wide and 120 tall. Stock at x 120-210, y 750-870. Waste at x 250-340, y 750-870.

7.6.2 Click Detection from Bottom to Top

We check array4 first. Loop its nodes with indices. Look up the x, y for that index. Test if click is within the rectangle. If yes, process that card and return. If no match, try array3. Then array2. Then array1. This bottom-to-top order respects visual layering since lower cards draw first.

7.7 Win and Loss Checking

7.7.1 Win Condition

Loop all 28 pyramid cards. If any has removed = false, return false. If all have removed = true, return true.

7.7.2 Loss Condition

First check if all pyramid cards are gone. If yes, game already won, return false. Check if stock has cards. If stock head is not null, return true. If stock is empty, loop all pyramid cards. For each visible non-removed card, extract rank and check match against waste. If any match, return true. Else return false.

7.8 Rendering Cards to Screen

Three states: removed (empty), hidden (blue back), visible (face front).

7.8.1 Removed Card State

Draw a faint dark rectangle. Add faint diagonal crossing lines. Nothing else. Shows where a card was.

7.8.2 Hidden Card State

Stack three blue rectangles offset for shadow. Rounded corners. Darker blue border. Circles in rows on the surface. Horizontal lines below circles. We draw 23+ hidden cards every frame at 60 FPS without lag.

7.8.3 Visible Card State

White rectangle with rounded corners and shadow. Suit color: red for hearts/diamonds, black for spades/clubs. Large rank text in center. Suit symbols in corners. Corner boxes matching suit color. Double line border for 3D effect. All calculated fresh every frame.

7.9 Why This Design Works

We use linked lists because appending to stacks happens constantly but random access never does. Multiple pointer types per node naturally express pyramid coverage without extra structures. Shuffling only the stock keeps initialization fast. The 52-card limit guarantees constant time operations. Real-time drawing works because card count is bounded. The whole system is efficient because the problem has an inherent size cap that never grows beyond what we can handle instantly.

8 Time Complexity

We needed to check if our game would be slow or fast. So we looked at what the code actually does and figured out the time complexity for each part. Basically we are using linked list that allow us to remain at time complexity of O(n), where n is the time taking to peform a function.

8.1 When Game Starts

8.1.1 `shuffle()` - Shuffling Cards

In easy mode we shuffle 5 cards. Medium is 10. Hard is 20. We're just swapping cards around. 5 swaps, 10 swaps, 20 swaps. Doesn't take long. The amount of cards to shuffle doesn't change based on anything. It's always the same number. So it's fast no matter what.

Time Complexity: O(1) - constant time. We shuffle a fixed number of cards every single time. No matter what difficulty, the number never grows.

8.1.2 `initializePyramid()` - Making the Pyramid

We take 28 cards and arrange them:

- 3 cards top
- 6 cards
- 9 cards
- 10 cards bottom

For each card we make a node and put it in a list. That's quick. 28 cards to process. Same every time. Very quick.

Time Complexity: O(1) - we always process exactly 28 cards, nothing more, nothing less. The pyramid size is locked in. So no matter what happens during the game, we do the same work at startup.

8.1.3 `initializeChildren()` - Linking Cards

Each card needs to know which cards are below it. So we go through and set up these links. 3 cards in top level link to stuff below. 6 cards in next level link. 9 cards. And so on. We do this once at start. Always same work. Always fast.

Time Complexity: O(1) - we're linking 28 cards to their children, but it's a fixed operation. The pyramid structure never changes size, so we always touch the exact same number of links.

8.2 While Playing

8.2.1 `handleClick()` - Finding What You Clicked

When you click, the code searches the pyramid to find which card you hit. It checks level 4 first (10 cards). Then level 3 (9 cards). Then level 2 (6 cards). Then level 1 (3 cards). Most we ever check is 28 cards. Just seeing if your click is on a card rectangle. Quick operation.

Time Complexity: O(1) - worst case we look through all 28 cards in the pyramid, but 28 is a constant. We never have more than 28 to search, so it doesn't get slower as you play.

8.2.2 `validateInput()` - Does Your Card Match?

Take the card you clicked. Compare it to the waste pile card. Are they one number apart? That's basically it. One comparison. Done.

Time Complexity: O(1) - just a single comparison between two numbers. Doesn't matter what cards exist, we always do exactly one check.

8.2.3 `revealCards()` - Show Hidden Cards

When you remove a card, some hidden cards become visible. We check the top 3 levels (18 cards total). For each one, we ask: are both cards below it gone? If yes, show this card. 18 cards to look at. Pretty fast.

Time Complexity: O(1) - we check a fixed 18 cards every time a card is removed. Not 28, not growing—always 18. Same work every time.

8.2.4 `drawFromStock()` - Get Next Card

When you click the stock pile, we grab one card and put it in the waste pile. Take the card. Create it. Add it. Done.

Time Complexity: O(1)- we're doing a few fixed operations: grab, create, add. Three simple steps that don't depend on how many cards are left. Always instant.

8.2.5 `checkWin()` - Did You Win?

Go through all 28 cards. Are they all removed? Yes or no. 28 cards. One check per card. Done.

Time Complexity: O(1) - we loop through 28 cards and check each one, but 28 is our constant. We never check more or fewer cards.

8.2.6 `checkLoss()` - Did You Lose?

Check if stock pile is empty. If yes, check all pyramid cards to see if any match the waste pile. 28 + some checks. Takes a moment but still instant.

Time Complexity: O(1) - worst case we compare the waste pile card against all 28 pyramid cards. But again, 28 is fixed. We never have more cards to compare.

8.3 What We Found

Thing	Function	Time Complexity	How Fast
Shuffle	`shuffle()`	O(1)	Pretty fast
Build pyramid	`initializePyramid()`	O(1)	Pretty fast
Link cards	`initializeChildren()`	O(1)	Pretty fast
Find your click	`handleClick()`	O(1)	Pretty fast
Check match	`validateInput()`	O(1)	Very fast
Show cards	`revealCards()`	O(1)	Pretty fast
Draw card	`drawFromStock()`	O(1)	Very fast
Check win	`checkWin()`	O(1)	Pretty fast
Check lose	`checkLoss()`	O(1)	Pretty fast

8.4 Why It's All Quick

We only have 52 cards. That's not a lot. Most of our checking is looking at 28 cards in the pyramid. That takes almost no time. Every single operation touches either a fixed small number of cards or just looks at specific pieces. Nothing gets worse as the game goes on. Everything stays the same speed from start to finish.

If we were dealing with thousands of cards, yeah maybe the math would look different. But we're not. So basically everything runs at **O(1)** constant time. You don't wait for anything. That's it. Game is fast because it's small and everything is constant-sized.

8.5 How We Figured This Out

We looked at each part of the code and asked: how many cards does this touch? Does that number change? Does it grow as we play?

`shuffle()` - always same amount. **O(1)**. Pyramid building - always 28 cards. **O(1)**. Checking cards - always max 28. **O(1)**. Finding matches - always one or a fixed set. **O(1)**. Nothing changes. Everything is constant. So everything is fast.

We didn't need to do complicated math. Just count the cards and figure out nothing gets worse as you play. Every function stays the same speed no matter what. That's why the whole game flies.

9 How We Used Linked Lists

9.1 Stock Pile and Waste Pile

These are the simplest linked lists in our game.

Stock pile (stack1):

- Starts with 23 cards
- When player clicks it, we remove the first card
- We add that card to the waste pile
- Then we delete the old node to free memory

Waste pile (stack2):

- Starts with 1 card
- Gets new cards added to the end
- We always look at the last card for matching

This is basic linked list work - adding to the end and removing from the front.

9.2 Pyramid Levels

Each level is a separate linked list.

- **Level 1:** linked list with 3 cards
- **Level 2:** linked list with 6 cards
- **Level 3:** linked list with 9 cards
- **Level 4:** linked list with 10 cards

We kept them separate instead of making one big list because:

- Easier to display each level
- Easier to find cards
- Simpler to set up children connections

When we need to find a card, we search through each list one by one.

9.3 Why Linked Lists Work Well Here

Advantages:

- Easy to add and remove cards
- Don't waste memory
- Good for small amounts of data like 52 cards
- Can easily connect cards with pointers (for pyramid structure)

Things that could be better:

- Can't jump directly to a card (must search from start)
- Uses extra memory for pointers
- But with only 52 cards, these don't matter much

10 Building the Pyramid Structure

10.1 Setting Up Four Levels

After you shuffle the deck, the bottom 28 cards are used to build the pyramid. You lay them out in four rows from top to bottom:

Level 1 (Top): 3 cards
Level 2: 6 cards
Level 3: 9 cards
Level 4 (Base): 10 cards

You basically stack each row so it overlaps the one above it, which forms the three-peak layout the game is named after.

Behind the scenes, each row is stored as a linked list. To fill a level, the program goes through the 28 cards one by one and adds each card to the end of that level's linked list in order.

10.2 Connecting Parents and Children

This was the hardest part. Each card in levels 1-3 must point to two cards below it.

For Level 3 to Level 4:

- Card 0 in Level 3 → Cards 0 and 1 in Level 4
- Card 1 in Level 3 → Cards 1 and 2 in Level 4
- Card 2 in Level 3 → Cards 2 and 3 in Level 4
- And so on...

For Level 2 to Level 3: This is more complex because of the three-peak pattern. We use a counter to track position and skip cards at certain spots to make the right shape.

For Level 1 to Level 2: Each card in Level 1 is placed above and center between two cards in Level 2, which naturally creates the spacing that forms the three peaks.

10.3 Where Cards Appear on Screen

In the visual version of TriPeaks, each card must be placed at an exact spot on the screen so that the layout lines up correctly. To do this, we use predefined lists of x-coordinates for each row (or level) of the pyramid. These lists tell the program exactly where to place each card from left to right.

Here are the x-coordinate lists:

Top row (3 cards): [300, 650, 1000]
Second row (6 cards): [230, 350, 580, 700, 930, 1050]
Third row (9 cards): [165, 285, 405, 525, 645, 765, 885, 1005, 1125]
Bottom row (10 cards): [110, 230, 350, 470, 590, 710, 830, 950, 1070, 1190]

These coordinates ensure the cards are evenly spaced, maintain the visual separation of the three peaks, and create the classic overlapping pyramid shape.

For vertical placement, each row is assigned a fixed y-coordinate so that every card in the same level aligns perfectly in height.

- Level 1: y = 160
- Level 2: y = 300
- Level 3: y = 440
- Level 4: y = 580

These numbers were chosen by testing to make the pyramid look good and centered.

11 How Player Moves Work

11.1 Clicking with the Mouse

When a player clicks, the program captures the mouse's x and y coordinates. The click-handling system follows a specific order:

Check the stock pile first: It verifies if the click happened within the stock pile's designated screen area.

Search the pyramid: If the click wasn't on the stock pile, it then checks the pyramid cards. The search starts from the bottom (Level 4) and moves upward (Level 3 → Level 2 → Level 1).

Card hit detection: For each card, the program checks if the mouse coordinates fall inside that card's rectangular bounds. Each card is defined as a rectangle 90 pixels wide and 120 pixels tall.

Example: A card positioned at (300, 160) occupies the area where:

x is between 300 and 390 (300 + 90) y is between 160 and 280 (160 + 120) A click within these coordinates would select that specific card.

11.2 What Happens After a Valid Click

When a player clicks a valid card:

1. We store which card was clicked
2. Check if the move is legal (using validateInput function)
3. If legal:
    - Mark the card as removed
    - Add 1 to the score
    - Add the card to the waste pile
    - Call reveal function to show any new cards
4. If not legal:
    - Nothing happens, player tries again

11.3 Drawing from Stock Pile

If player clicks the stock pile:

1. Check if any cards are left in stock pile
2. If yes:
    - Remove first card from stock pile linked list
    - Create new node with that card's data
    - Add new node to end of waste pile
    - Delete the old node from memory
3. If no:
    - Nothing happens

This gives the player a new card to match with, but doesn't increase their score.

12 Difficulty Levels

12.1 Choosing Difficulty

Before the game starts, the player sees three buttons:

- **Easy (Green button):**
    - Shuffles 5 cards
    - Easiest to win
- **Medium (Orange button):**
    - Shuffles 10 cards
    - Medium difficulty
- **Hard (Red button):**
    - Shuffles 20 cards
    - Hardest to win

When player clicks a button, we save their choice and start the game.

12.2 How to make game levels:

This is done by function that how many cards will be shuffled.

Easy mode:

- Most cards stay in order
- Game is more predictable
- Good for learning

Medium mode:

- More randomness
- Need to think ahead more

Hard mode:

- Very random
- Almost no predictable cards
- Requires luck and skill

The rest of the game works exactly the same for all difficulties.

13 Graphics and Display

13.1 Drawing Cards

Each card is a rounded rectangle. We use Raylib's drawing functions.

Card states:

1. **Removed cards:**
   - Very light outline
   - Almost invisible
   - Shows where card used to be
2. **Hidden cards:**
   - Blue background
   - Pattern of circles on it
   - Looks like back of playing card
3. **Revealed cards:**
   - White background
   - Shows card name (like SA, H7)
   - Red color for Hearts and Diamonds

- Black color for Spades and Clubs
- Small shadow behind card

Card size:

- Width: 90 pixels
- Height: 120 pixels
- Rounded corners for nice look

13.2 Screen Layout

Score panel (top left):

- Black background with gold border
- Shows current score
- Shows chosen difficulty

Pyramid (center):

- Cards arranged in four rows
- Each row is a different level
- Creates three-peak pattern

Stock and waste piles (bottom):

- Stock pile on left side
- Waste pile next to it
- Both at y = 750

Colors:

- Background: Forest green (like a card table)
- Cards: White when revealed, blue when hidden
- Text: White, yellow, or gold for good contrast

13.3 Different Screens

The game has three screens:

1. **Difficulty Selection Screen:**
   - Shows title "TRIPEAKS SOLITAIRE"
   - Three colored buttons
   - Instructions for player
2. **Game Playing Screen:**
   - Shows pyramid
   - Shows stock and waste piles
   - Shows score
   - Player can click cards
3. **Game Over Screen:**
   - Dark overlay on game
   - Big text saying "YOU WIN" or "GAME OVER"
   - Final score
   - To restart, press R or to quit, press ESC.

14 Problems occured during this project:

14.1 Parent-Child Connections Were Hard

Problem: Setting up which card covers which other cards was confusing. The three peaks overlap in a specific way that's hard to calculate.

Solution: We drew the pyramid on paper first. We numbered each position. Then we figured out the pattern by looking at our drawing. We wrote the code step by step, testing each level's connections separately.

14.2 Memory Problems

Problem: When we removed cards from the stock pile, we created new nodes but forgot to delete old ones. This wasted memory.

Solution: We added "delete" commands wherever we removed a card from a linked list. We checked for memory leaks and fixed them.

14.3 Finding Right Positions for Cards

Problem: Getting cards to line up nicely on screen took many tries. We had to guess x and y positions.

Solution: We started with approximate positions. Then we ran the game, looked at it, and adjusted numbers. We did this many times until it looked good. We wrote down the final positions in arrays so we wouldn't lose them.

14.4 Click Detection Issues

Problem: Sometimes clicking a card didn't work. The click hit the wrong card or no card.

Solution: We made the program print out where the mouse clicked. This helped us see if our rectangle calculations were wrong. We fixed the boundaries and tested by clicking all over the screen.

15 Testing the Game

15.1 Testing Console Version

We tested each function separately first.

Shuffle testing:

- Printed cards before shuffle
- Printed cards after shuffle
- Checked that right number of cards moved

Pyramid setup testing:

- Printed each level's cards
- Counted cards in each level
- Made sure total was 28

Parent-child testing:

- Printed pointer addresses
- Checked if they matched what we expected
- Drew diagrams to compare

Move validation testing:

- Tried moves we knew should work
- Tried moves we knew should fail
- Tested edge cases like King-Ace matching

15.2 Testing GUI Version

Mouse click testing:

- Clicked on each card position
- Made sure right card was selected
- Checked boundary areas between cards

Visual testing:

- Looked at game on different monitors
- Checked if colors were easy to see
- Made sure text was readable

Button testing:

- Clicked each difficulty button
- Verified right difficulty was chosen
- Checked that game started properly

15.3 Playing Complete Games

We played the game many times to test everything together.

Win testing:

- Played games until we cleared pyramid
- Checked that "YOU WIN" appeared
- Verified score was correct

Loss testing:

- Played until stuck on purpose
- Made sure "GAME OVER" appeared
- Tested restart function

Score testing:

- Watched score go up with each card
- Made sure it reset when restarting
- Checked it showed correct final score

16 What We Learned

16.1 About Linked Lists

We learned that linked lists are good for this type of game because:

- Cards are always being added and removed
- We don't need random access to middle cards
- The pyramid structure works well with child pointers

We also learned that:

- Memory management is important
- Must remember to delete unused nodes
- Pointers can connect data in complex ways

16.2 About Building Games

Start simple: Console version first made everything easier. We could test logic without worrying about graphics.

Test small pieces: Testing each function separately found bugs early. Fixing small problems is easier than fixing big problems.

Plan the structure: Drawing the pyramid on paper before coding helped a lot. Understanding the structure first made coding faster.

17 Conclusion

We successfully made a TriPeaks Solitaire game using linked lists. The game works well and is fun to play.

What we accomplished:

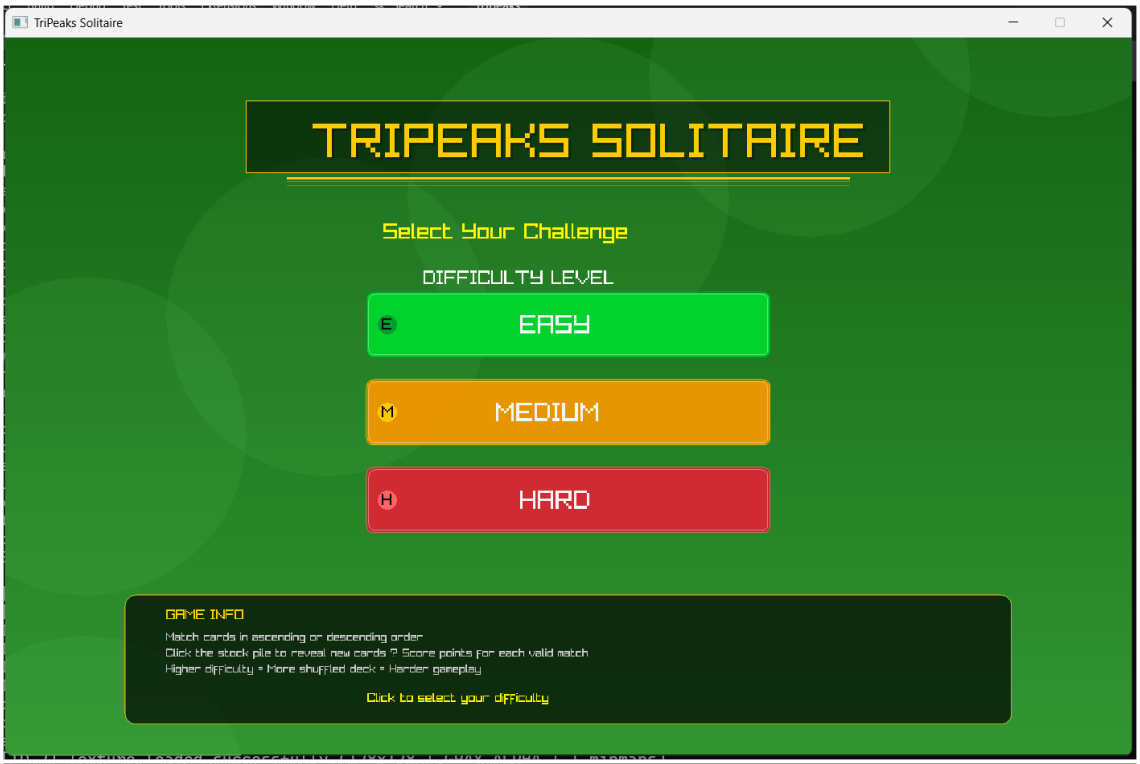
- Built working console version
- Added nice graphics with Raylib
- Implemented all game rules correctly
- Created three difficulty levels
- Made it easy to play with mouse clicks

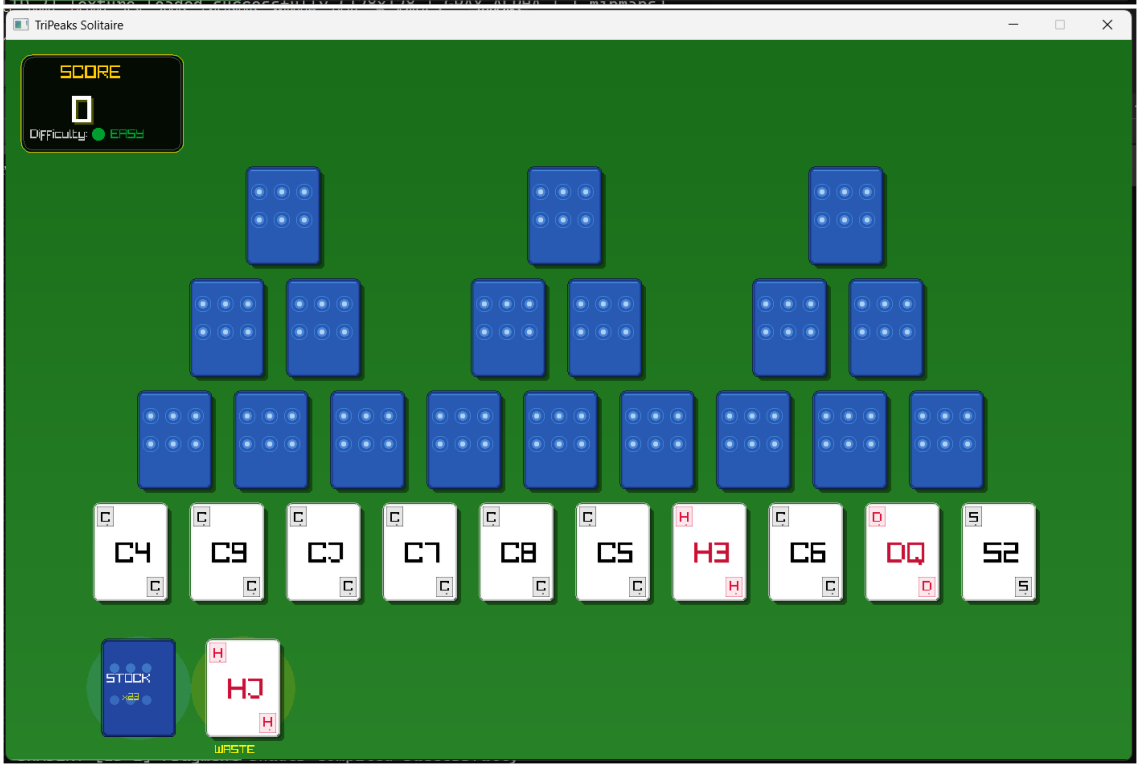
How linked lists helped:

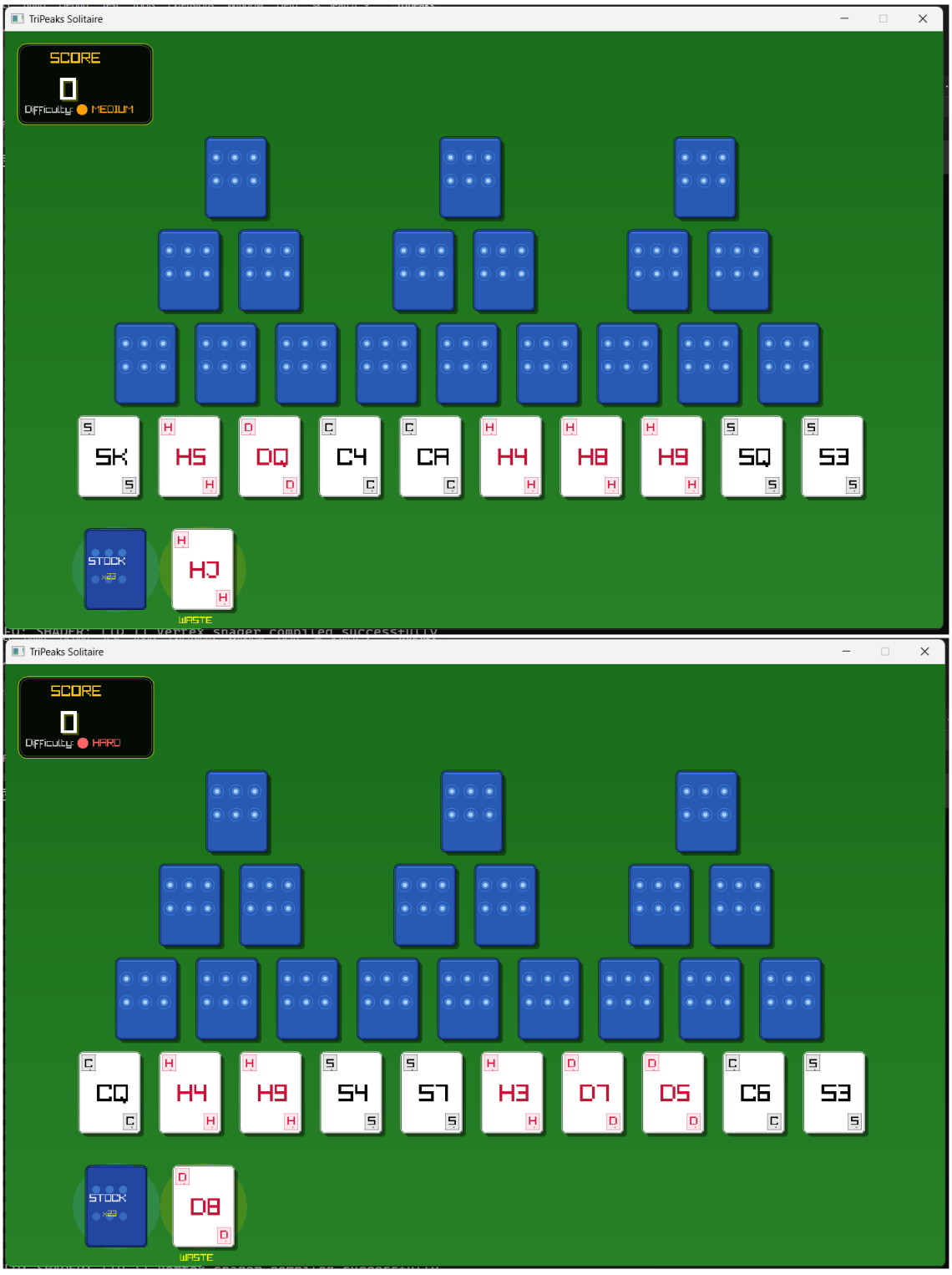
- Good for managing card piles
- Easy to add and remove cards
- Child pointers made pyramid structure work
- Simple to understand and debug

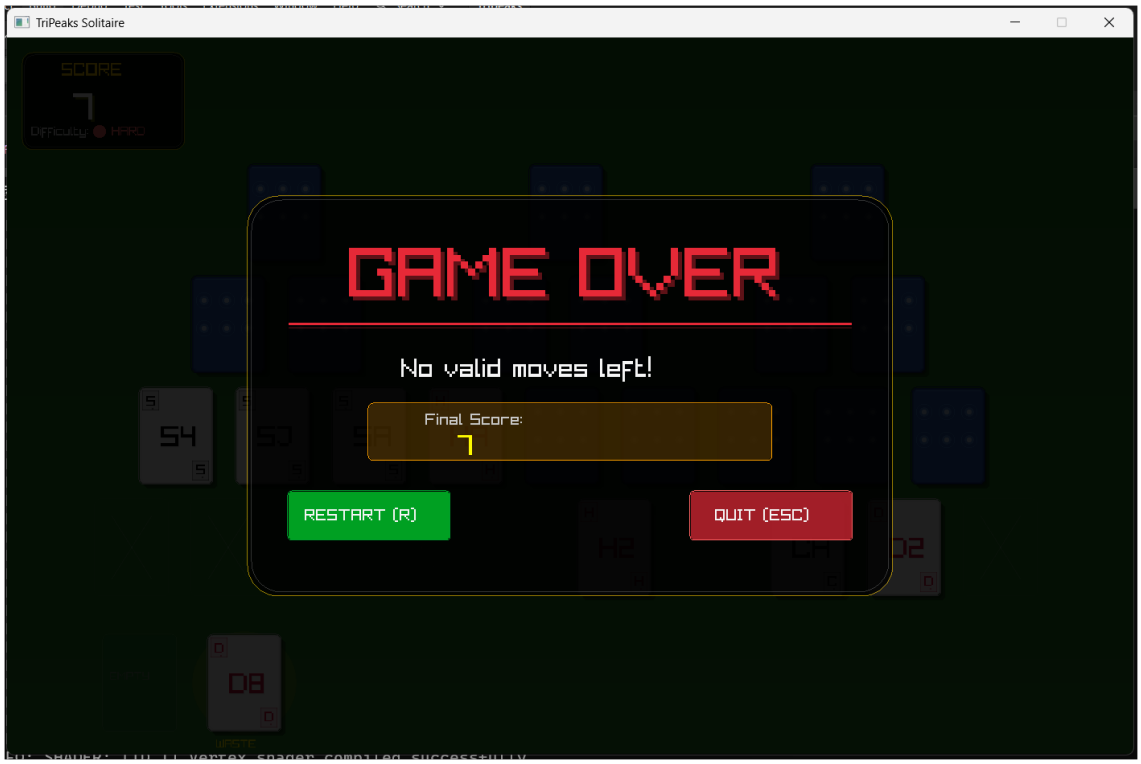
What could be added later:

- Undo button (could use another linked list to store previous moves)
- Animations when cards move
- Sound effects
- High score saving
- Hint system to show possible moves


TriPeaks Solitaire
TRIPEAKS SOLITAIRE
Select Your Challenge
DIFFICULTY LEVEL
E
EASY
M
MEDIUM
H
HARD
GAME INFO
Match cards in ascending or descending order
Click the stock pile to reveal new cards ? Score points for each valid match
Higher difficulty = More shuffled deck = Harder gameplay
Click to select your difficulty


TriPeaks Solitaire
SCORE
0
Difficulty: EASY
C4
C9
CJ
C7
C8
C5
H3
C6
DQ
S2
STOCK
HJ
WASTE


TriPeaks Solitaire
SCORE
0
Difficulty: MEDIUM
SK
H5
DQ
C4
CA
H4
H8
H9
SQ
S3
STOCK
HJ
WASTE
TriPeaks Solitaire
SCORE
0
Difficulty: HARD
CQ
H4
H9
S4
S7
H3
D7
D5
C6
S3
STOCK
D8
WASTE


TriPeaks Solitaire
SCORE
7
Difficulty HARD
GAME OVER
No valid moves left!
Final Score:
7
RESTART (R)
QUIT (ESC)