

## **Project 2**

**Title: WAR**  
**A card game**

**Course:**  
**CSC-17C**

**Section:**  
**48881**

**Due Date:**  
**Dec 17<sup>th</sup>, 2023**

**Author:**  
**Michael Donnelly**

## Table of Contents

Pg 1 Cover

Pg 2 Table of Contents

Pg 3 Introduction

Pg 4 Approach to Development

Pg 5 Game Rules

Pg 6 Description of Code

Pg 7 Menu #1 Play Game – **RECURSIVE SORT**

Pg 8 Menu #2 Stats – **GRAPH**

Pg 9 Menu #3 Stats all – **HASHING**

Pg 10 Menu #4 Previous Menu Choices

Pg 11 Menu #5 Test Recursive Function – **RECURSIVE FUNCTION**

Pg 12 Menu #6 Exit game – **TREE**

Pg 13 Project #2 Checklist

Pg 14 Memes

## Introduction

I am coding the card game “WAR” with a x4 card draw variant. This was the game I played in elementary school and so I have been thinking about this game since then. I always wondered if there was a way to know who would win before the game got played, but the problem was finding other willing participants to play WAR against you. People stop wanting to play this game because it isn’t necessarily fun and it’s not fun because there is ZERO user input. The epiphany I had then was that WAR didn’t require another player and that I could play a game against myself. My program accomplishes this and then simulates how much time it would have taken you to run the game yourself with your own deck of cards so you can see how much time you avoided wasting by using my program.

I spent an equal amount of time planning this project as I did coding for it: Altogether, I spent about 20 hours planning, designing, and coding it. The most challenging part that took the longest was designing the scope of the game such that it would meet the project requirements. What I planned out did not meet the project length requirement so in the future I would pay special attention to this until I learn to better estimate how much code it will take to make each section of a project

Lines of Code	>750
Whitespace	>62
comments	>153
Total Lines	1230

This game has 3 class.h files. One stores and outputs terrible ASCII art and the other 2 are for the AVL Tree implementation.

Github link to project:

[https://github.com/Marnatee/CSC17C/tree/main/CSC17C\\_Project2v1](https://github.com/Marnatee/CSC17C/tree/main/CSC17C_Project2v1)

## **Approach to Development**

**Concepts:** Originally, I was disappointed about not being able to use Vectors for this project but then when I thought about the game of WAR, it consists of 2 phases, skirmishes and WARs. Skirmishes take the top card of the player's hand and winnings are deposited at the bottom of the player's hand which is EXACTLY like a Queue container. WARs are done by having the player stack 4 cards with the 4th card placed faceup which is EXACTLY like a Stack container. I can't use Vectors but Queues and Stacks were all I ended up needing. The challenge is making sure the program does the right checks to make sure you have enough cards to enter WAR.

The portion of the program I am most proud of is how I got the WAR() function to call itself recursively. I thought originally that I'd have to code every single WAR scenario but with recursion, I did not have to. I only hard coded the function to detect up to a quad (x4) war but it will still run properly with a quintuple war or higher.

**Version Control:** I only have one version of the program but I did extensive planning of my program on paper and a whiteboard before I ever committed anything to Netbeans.

### **Game Rules:**

A deck of 52 cards is shuffled and then each player gets 26 cards.

Each player must play their top card:

If your card is greater than your opponent's, you take both cards and place at bottom of your Hand.

Vice versa if opponent's card is greater.

If both cards are equal, then you go to WAR.

For WAR, each player makes a stack on their prior card with 3 cards facedown and one more card faceup.

if your faceup card is greater than your opponent's, you claim the entire pile and return it to the bottom of your hand. Vice versa if opponent's card is greater.

If both cards are equal, then you go to DOUBLE WAR and repeat the process.

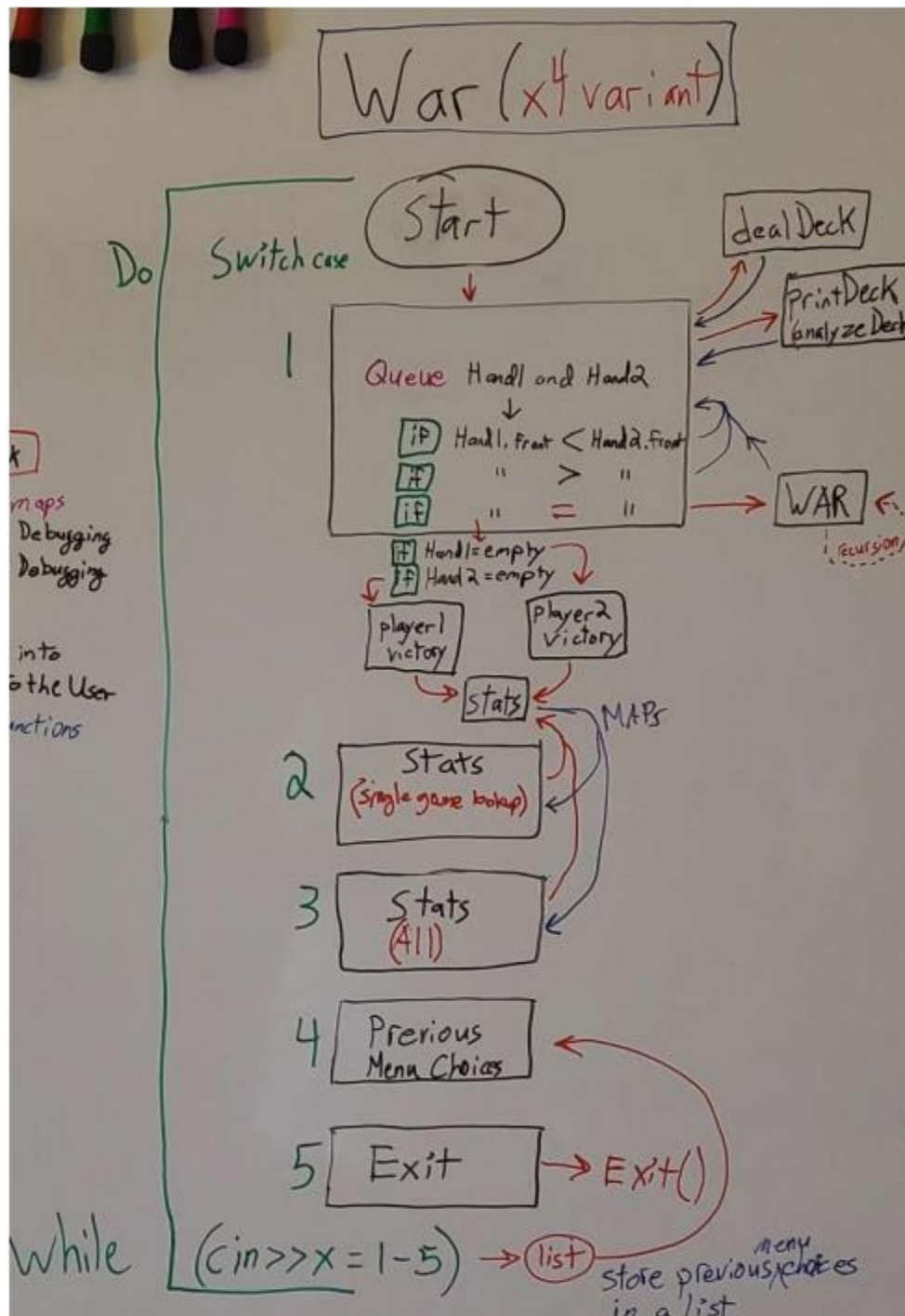
NOTE: This can lead to TRIPLE WAR / QUAD WAR / QUINTUPLE WAR or greater if unlucky enough.

This continues until one player no longer has any cards in their hand. Once this condition occurs, the other player wins.

**NOTE:** According to [bicyclecards.com](http://bicyclecards.com) each player should only have 13 cards (Mine = 26 each) and only 1 card is laid facedown during WARs (Mine = 3 cards facedown). Because of this, I have named my game the "x4 variant

## Description of Code:

This is a snippet from my Whiteboard. Full image is in the project folder.



The program starts by entering a Switch case (not actually a switch case, but it's setup like one) with 1 = play game / 2 = Stats (single game lookup) / 3 = Stats (all games) / 4=See previous menu choices / 5=Exit program. The switch is initialized to 1 such that the program will automatically play a single game of WAR when you run it.

## 1) Play Game

The program starts by creating an array Deck with 52 elements and then calls the dealDeck() function which uses a Set {1,2,3,...13} to initialize the deck to have 4 of each card value 1<->13 (11,12,13 are face cards J/Q/K). This deck is now shuffled twice with the knuth\_b protocol. The copy() algorithm is used to copy this deck and then the sort() algorithm is used to sort it. Both unsorted and sorted decks are then displayed to the user so that the user can verify that the cards were dealt right.

Next the program feeds the randomized Deck into the prntDeck() function which prints out the first 26 elements as the Player's hand and the last 26 elements as the CPU's hand. The player's hand is then analyzed in two ways. First, the Hand is summed up and compared to the worst possible hand (98 = 1,1,1,1,2,2,2,3,3,3,3,4,4,4,5,5,5,6,6,6,7,7) and the best possible hand (266 = 7,7,8,8,8,8,9,9,9,10,10,10,10,11,11,11,11,12,12,12,12,13,13,13,13). Your hand is given a percentage ranking of where it stands in the range of 98-266. Second, the count() algorithm is used to count how many face cards (J/Q/K) are in your hand since they are powerful cards in WAR and almost always beat the card they are played against.

## SKIRMISHES

Next the program creates 2 queues: Hand1 (first 26 elements of Deck) and Hand2 (last 26 elements of Deck) proceeds on to skirmishes where the front card of each Queue is compared against each other. If your card is greater than your opponent's card, you win both cards and push them into your Queue at the back. If your opponent's card is greater, then the opposite happens. If both cards are greater then the WAR condition triggers and the WAR() function is called.

```
P1 Hand (22 cards): 8 7 K 9 J 4 3 J 4 8 J 5 8 Q 6 6 K 3 7 K 10 1
P2 Hand (30 cards): 9 10 J 9 5 4 10 5 Q 2 1 5 6 Q K 3 1 2 1 Q 10 9 6 8 2 3 7 7 4 2
Back to Skirmishes!
8 vs 9 7 vs 10 K vs J 9 vs 9
WAR!
```

## WAR

The previous 2 cards + 4 additional cards each are taken from each player's Queue and added to a Stack called WAR1 and WAR2. The top card of each Stack is then compared against each other the same way the Queues were compared in Skirmishes. If your card is greater, you claim both stacks of cards and put them at the back of your queue. If the cards are equal, then the WAR() function calls itself RECURSIVELY and adds 4 more cards to each stack and does another comparison until it solves it.

The card comparisons are continued with skirmishes and WARs until one player's Hand runs out of cards triggering the victory condition for the other Player. Game stats are then added to multiple Map containers.

## 2) Stats (single game lookup) – GRAPH implementation

The program asks the user for a specific game to lookup and then will re-output all of the stats from that game. All data is stored in Maps, so this will print out the Maps in the user-selected range.

```
MENU - Choose input 1-6
(last option picked = 55)
1 = play again
2 = Stats (specific match)
3 = Stats (all matches)
4 = See previous menu choices
5 = Test recursive function <-- (TRY ME)
6 = EXIT (Plant a tree)
2

Input value from 1 to 2
2

GAME 2 LOSS
Total skirmishes = 0
Player 1 skirmishes won = 0
Wars fought = 2
Wars won = 0
Double wars = 1
Triple wars = 1
Quad wars = 1
Node 0 makes an edge with
Node 1 makes an edge with
    Node 2 with edge weight = 0
game #2
Node 2 makes an edge with
    Node 1 with edge weight = 0
    Node 2 with edge weight = 0
    Node 2 with edge weight = 0
    Node 3 with edge weight = 2
    Node 8 with edge weight = 0
    Node 4 with edge weight = 1
    Node 5 with edge weight = 1
    Node 6 with edge weight = 1
```

**data  
grabbed**

**same  
data as  
graph**



### 3) Stats (all games) – HASHING implementation

The program will output all game stats, listed game by game. At the end it will output a TOTAL STATS block that tells you how many games you won out of total games played / skirmishes won out of total skirmishes / wars won out of total wars fought / how many double triple quad wars you got (if any) / The total simulated play time

```
SSSSS  TTTTT      A      TTTTT  SSSSS
S       T       A A      T       S
SSSSS   T      AAAAA   T      SSSSS
      S   T      A      A      T      S
SSSSS   T      A      A      T      SSSSS
```

GAME 1 WIN

Total skirmishes = 133

Player 1 skirmishes won = 72

Wars fought = 9

Wars won = 6

Double wars = 0

Triple wars = 0

Quad wars = 0

Simulated Play Time = 4min 13sec

**hashed value**

TOTAL STATS FROM ALL 1 GAMES

You won 1 games out of 1 total

You won 72 skirmishes out of 133 total

You won 6 WARs out of 9 total

No Double Wars

No Triple Wars

No Quad Wars

Key 253 found: 1

**hash retrieved**

**hash  
used**

Total Simulated Play Time = 4min 13sec

#### 4) Previous Menu Choices

This menu option uses a List container to store every choice the player makes in the menu in order. The element at the very back is also displayed on the menu to tell you what you chose last.

```
You selected these choices
```

```
1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 4,
```

I played 3 games, then printed all my game stats, then played 16 more games, then printed stats, then viewed my menu choices.

## 5) Test Recursive Function – RECURSIVE function implementation

This menu option injects a Trick Deck into the player and Ai's hands which forces a Sextuple (x6) war to occur. The WAR() function is called and then calls itself recursively 5 times.

```
Player 1's Hand (YOU)
K 9 9 8 K 8 7 7 Q 6 6 5 Q 5 4 4 J 3 3
2 J 2 1 1 10 10
Player 2's Hand (AI)
K 9 9 8 K 8 7 7 Q 6 6 5 Q 5 4 4 J 3 3
2 J 2 1 1 10 10
Quadrouple war has been declared!
  4 4
  4 4
x x 4444
  x 4
x x 4
```

 **trick deck is injected into player / ai hands**

```
Each player lays down 4 more cards again for a total of 32+2 cards
Player 1 top card = J
Player 2 top card = J
```


```
STALEMATE!
ANOTHER WAR!
```

```
Quintuple war has been declared!
  5555
  5
x x 5555
  x 5
x x 5555
```

```
Each player lays down 4 more cards again for a total of 40+2 cards
Player 1 top card = J
Player 2 top card = J
```

```
STALEMATE!
ANOTHER WAR!
```

```
Sextuple war has been declared!
  6
  6
x x 6666
  x 6 6
x x 6666
```

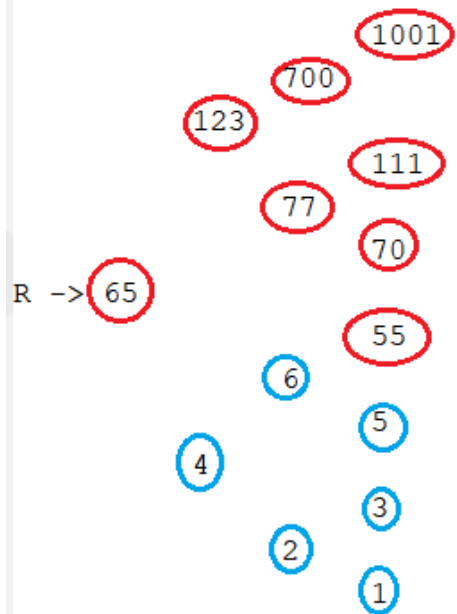
 **trick deck is primed to trigger sextuple war condition**

```
Each player lays down 4 more cards again for a total of 48+2 cards
WAR() function has called itself recursively 5 times
```

## 6) Exit (and plant a tree) – TREE implementation

This menu option exits the game but before it does, it takes the linked list from #4 and inputs it into a Tree and then displays the tree horizontally with every unique menu choice that was made by the user (this includes erroneous choices).

Creating and displaying a tree with all (unique) menu options visited



**These aren't actual menu options, they are erroneous values inputted by the user when they were asked to input 1-6**

**menu values 1-6 won't display here unless the user visits it**

```
Exiting program. Goodbye.
EEEE  X   X   I I I I I   T T T T T
E      X X     I         T
EEE    X       I         T
E      X X     I         T
EEEE  X   X   I I I I I   T
```

## **Project 2 Checklist**

### **Recursion**

Use menu option #5 to test this  
called on [line 232](#)  
implementation = [line 565-743](#)

### **Recursive Sort**

Use menu option #1 to test this.  
I used QuickSort and replaced STL Sort() with it on [line 772](#)  
Implementation = [line 857-897](#)

### **Hashing**

Use menu option #3 to test this.  
I add the simulated game time to a hash table and then look up the stored value  
used on [lines 425-431](#)  
Implementation = [line 56-93](#)

### **Trees**

Use menu option #6 to test this.  
used on [lines 460-469](#)  
Implementation is in [AVLTree.h](#) and [BNTnode.h](#)

### **Graphs**

Use menu option #2 to test this.  
used on [lines 361-370](#) to duplicate game stats displayed  
Implementation on [lines 25-54](#)

## Memes



```
if (sizeSTACK==5)w2+=1; //double war counter
if (sizeSTACK==9)w3+=1; //triple war counter
if (sizeSTACK==13)w4+=1; //quad war counter
```

**3 lines**



```
if (sizeSTACK==5){
    w2+=1; //double war counter
}
if (sizeSTACK==9){
    w3+=1; //triple war counter
}
if (sizeSTACK==13){
    w4+=1; //quad war counter
}
```

**9 lines**

```
if (sizeSTACK==5)w2+=1;
if (sizeSTACK==9)w3+=1;
if (sizeSTACK==13)w4+=1;
```

**3 lines**

```
if (sizeSTACK==5){
    w2+=1; //double war counter
}
if (sizeSTACK==9){
    w3+=1; //triple war counter
}
if (sizeSTACK==13){
    w4+=1; //quad war counter
}
```

**9 lines**

```
switch (sizeSTACK){
    case 5:
        w2+=1; //double war counter
        break;
    case 9:
        w3+=1; //triple war counter
        break;
    case 13:
        w4+=1; //quad war counter
        break;
}
```

**11 lines**

