Connect Four

Algorithms and Data Structures coursework report

Author: Klaudia Jaros 40439268

Contents

[Introduction 2](#_Toc70187444)

[Design 2](#_Toc70187445)

[User Interface 2](#_Toc70187446)

[Data Structures 2](#_Toc70187447)

[Algorithm to check for winners 3](#_Toc70187448)

[Gameplay algorithm 4](#_Toc70187449)

[Single player mode 4](#_Toc70187450)

[Assisted mode 4](#_Toc70187451)

[General design 5](#_Toc70187452)

[Critical Evaluation 5](#_Toc70187453)

[References 6](#_Toc70187454)

[Appendices 7](#_Toc70187455)

# Introduction

Connect Four is a two-player game where two players take turns dropping distinctive discs into a seven-column, six-row grid. The objective of the game is to be the first to form a horizontal, vertical, or diagonal connection of four of one's own discs ([Wolfram Math World, 2021](#_References)).

In this coursework, I have implemented Connect Four as a command line application using the C programming language, C standard libraries and self-written libraries. The game features two play modes: single player – a user against a computer (where moves are randomly generated) and a multiplayer mode – two users playing together. Both modes can be played using assisted mode that allows players to undo and redo moves, or alternatively in a competitive mode that does not include this feature. Additionally, any game played during the application runtime is being saved to allow for game replays. Past games can be found in the “Game History” option of the main menu and can be re-watched. The size of the board can also be adjusted to accommodate the most common variations: 5×4, 6×5, 8×7, 9×7, 10×7, 8×8.

# Design

## User Interface

As mentioned above, Connect Four is a command line application therefore a simple Text User Interface has been implemented to provide the basic functionalities ([Appendix](#_Appendices) B).

## Data Structures

To allow the game to function correctly, several data structures were needed to store information about the game, players and moves.

1. **Game board**

Game board was implemented using a simple *char array*. The array stores either ‘X’ or ‘O’ as the players’ moves or an empty space if not occupied by any move. The reason for using an array is that a fixed size of the board is provided by the user at the beginning of the program, and because an array allows for an easy direct access using index and offset to save players’ moves. It is also easy to print the entire board for the user to see and it does not require a lot of memory.

1. **Players**

A *Player struct* was used to store information about the players such as their names, token ( ‘X’ or ‘O’ ) and flags (one indicating for a “computer” player, and another to indicate the winner of the game). This data structure was used because it allows for grouping all information together despite them being of a different type([Olsson, 2019, p. 63](#_References)), and is easily accessible throughout the game play. A pointer to a struct can also be easily saved to Game History and then used to replay the game using players’ names.

1. **Saving game moves**

To easily save players’ moves and undo/redo them, a *stack structure* was needed where items can be added and deleted only at the end of the list. It was implemented as a struct with an array and an integer to keep track of the top(end) of the stack ([Nielsen, 2009, p. 186](#_References)). In assisted mode where the players can undo and redo their moves, two stacks were used: one for players’ moves and one for storing undone moves to support the redo moves option. Stacks allow for an easy pop and push operations on the last element of the list which is very useful when dealing with undoing moves or “going back”.

Because the stack struct still uses an array, it was possible to turn it into a *queue* for the purposes of re-watching a past game. Instead of using pop and push, a simple loop was set that starts at the beginning of the array and replays the players’ moves following a first in first out order, rather than stack’s way of last in first out.

1. **Saving games history**

To display an unspecified number of past games to replay them, another list was needed. A *Linked**List* logic ([Adamson, 1996, p. 13](#_References)) was used to create a GameHistory struct that stores a list of moves that were played in a game(a pointer to a Stack struct), two players (pointers to Player structs), board size (two integers: columns and rows), and a pointer to the next GameHistory struct (the link part of a linked list). Because different data structures were needed to be stored within that list and its size is not constant, a simple array would not work in this case. A Linked List was the best option - it allows for dynamic insertion to add more games, and when listing all the saved games, the starting point is at the beginning of the list and then it follows to the next link until it reaches a null link.

## Algorithm to check for winners

After every turn, the program needs to check for possible winners. To achieve that, a modified*linear search* algorithm with a counter was deployed to check the char array (board) one element at a time for a horizontal, vertical, or diagonal connection of four of a kind. If the element repeats, the counter is incremented. If the next element is the opposite element, the counter gets reset. If the counter reaches 4, the game is over. In the game source code, checkWin() method is implementing this logic.

To check for winners horizontally, a simple loop was used to move from one cell to another, resetting the counter at the beginning of a new row. To check it vertically, two loops were needed: one to start at the bottom of each column and a second one to move up. An offset was needed to move from a cell at the bottom of one column to a cell up in the same column.

To check for winners diagonally down, one nested loop was needed to check the first half of the board starting from the beginning of each row and moving diagonally down by adding an offset, and a second nested loop to check the second half of the board starting at the top of each column and moving diagonally down again – see the table below.

**Table 1: Check diagonally down - moving from the start point and going diagonally down by adding an offset of 8 in the 7x6 board**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Start point | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Start point | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 | 32 | 33 | 34 |
| 35 | 36 | 37 | 38 | 39 | 40 | 41 |

To check diagonally up, the same approach was used as above, but instead of moving down at each iteration, the loop moves up. Also, for the second half of the board, it starts at the bottom of each column and it is moving up – see table below.

**Table 2: Check diagonally up - moving from the start point and going diagonally up by subtracting 6 in the 7x6 board**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Start point | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 | 32 | 33 | 34 |
|  | 35 | 36 | 37 | 38 | 39 | 40 | 41 |
| Start point | | | | | | |

The above tables are illustrative. ConnectFour supports different board sizes, therefore the offset is not hardcoded, it is calculated based on the user’s choice of board size ([Appendix C](#_Appendix)).

## Gameplay algorithm

ConnectFour application consists of several loops and decisions based on the user’s input. The entire program logic can be seen on a Connect Four Flowchart in [Appendix](#_Appendix) A.

## Single player mode

A “computer” opponent was required to allow for a single player game. To achieve that, the Player struct was equipped with a flag to indicate whether the next player is a “computer”. If yes, instead of allowing a human player to take turn, a random number generator was used to generate a number between 1 and the maximum column number. If the generated move is invalid, the random number generator will try again and so on until a valid move is played ([Appendix](#_Appendix) A).

## Assisted mode

Every move is being saved in the moves stack. The undo/redo move option in the assisted mode utilises stack properties: if the player undoes their move, the move stack pops an element and the undone moves stack pushes one. If the user redoes their move, the undone moves stack pops an element, and the moves stack pushes one.

In the multiplayer assisted mode, each user can only undo/redo their own moves and the moves have to stay balanced during the gameplay.

In the single player assisted mode, this is slightly different. The player can undo both their own moves as well as the computer’s. This was implemented because otherwise it would have been impossible to go back to the beginning of the game given the computer cannot undo its own moves. Also, if the player is still learning how to play, it is more convenient to be able to undo the computer’s moves.

## General design

The Stack struct, Game History struct, and their methods have been placed into separate source files and compiled into a library to then be linked with the main application to keep the code readable and maintainable.

The main application has also been divided into small functions to make it more concise and easier to modify. Depending on the user’s choice throughout gameplay, the correct method is called.

# Critical Evaluation

Overall, this Connect Four game is fully functioning ([Appendix D](#_Appendix)). Features that work well are the single player and multiplayer modes, saving user’s moves, displaying the game history, and replaying past games. Undo and redo options work well too.

Things that could be improved include introducing different difficulty settings in the single player mode. Currently the “computer” uses randomly generated moves which are rarely challenging. The “computer moves” are randomly chosen between 1 and the maximum column size, which could result in the computer trying to take a turn in an already full column. Fortunately, the loop will generate a new number if this happens. In cases where moves are limited, it can take 2-3 seconds to generate a new valid move, which is noticeable by the player. In place of this, a list of valid moves could be implemented for the computer to choose from, avoiding invalid computer moves.

# References

**Adamson, I. T. (1996) *Data Structures and Algorithms: A First Course*. Springer**

**Kernighan, B. W. & Ritchie, D. M. (1988) *The C programming language* . 2nd ed. Englewood Cliffs, N.J: Prentice Hall.**

**Nielsen F. (2009) *A Concise and Practical Introduction to Programming Algorithms in Java*. Springer**

**Olsson M. (2019) *Modern C Quick Syntax Reference* . 2nd ed. Apress**

**Wolfram Math World (2021), *Connect Four*, viewed on 22 March 2021,** **https://mathworld.wolfram.com/Connect-Four.html**

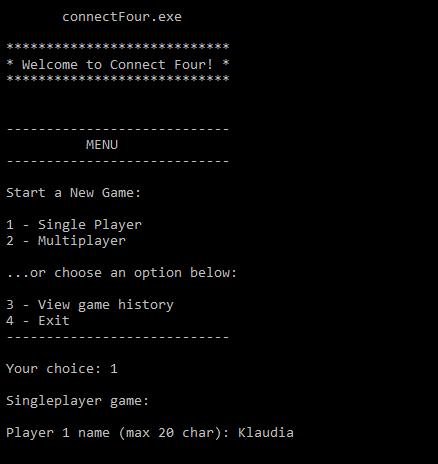
# Appendices

**Appendix A - Connect Four application Flowchart**

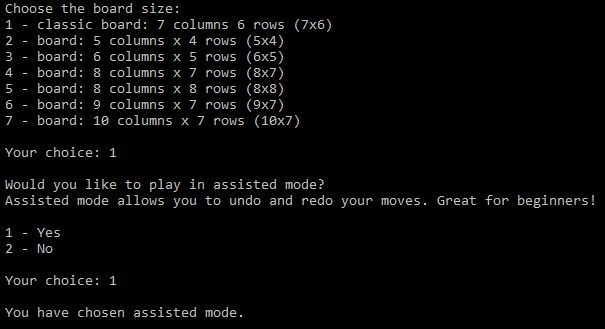
Diagram

Description automatically generated

**Appendix B – Screenshot of the Text User Interface designed for Connect Four**

****

**Appendix C – Screenshot of choosing the board size and assisted mode game**



**Appendix D – Screenshot of the end of a single player game in assisted mode (left) and a competitive multiplayer (right)**

