Connect Four

Algorithms and Data Structures coursework report

Author: Klaudia Jaros 40439268

Contents

[Introduction 1](#_Toc69224180)

[Design 1](#_Toc69224181)

[Critical Evaluation 1](#_Toc69224182)

[References 1](#_Toc69224183)

# Introduction

“Connect Four is a two-player connection board game, in which the players choose a colour and then take turns dropping coloured discs into a seven-column, six-row vertically suspended grid. The pieces fall straight down, occupying the lowest available space within the column. The objective of the game is to be the first to form a horizontal, vertical, or diagonal line of four of one's own discs” ([Wikipedia, 2021](#_References)).

In this coursework, an attempt has been made to implement Connect Four game as a command line application using the C programming language, C standard libraries and self-written libraries. The result of the design and implementation of the above idea is a working protype of the game Connect Four, featuring two play modes: single player – a user against a simple randomly generated computer moves and a multiplayer – two users playing together. Both modes can be played using the assisted mode that allows players to undo and redo moves or in a competitive mode without allowing the users to undo their moves. Additionally, any game played during the application runtime is being saved to allow for game replays. Past games can be found the the “Game History” in the main menu and can be re-watched.

# Design

## User Interface

As mentioned above, Connect Four is a command line application therefore a simple Text User Interface has been implemented.

1. **Screenshot of the TUI**

The application has been divided into small functions to make it more readable and less confusing. Depending on the user’s choice, the right method will be called.

## 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 of size 42 since the board is 6x7 in size. The array stores either ‘X’ or ‘O’ as the player’s moves or ‘ ‘ an empty space if not occupied by the user’s move. The reason for using an array is that a specified size was used for the board, it is easy to mark where the move was made, and it is easy to print the entire board for the user to see.

1. **Players**

To store information about the players such as their names, token ( ‘X’ or ‘O’ ) and flags if the player is a “computer” for the single player mode or if the player is the winner a struct was used. The reason for that is that a struct very nicely groups all the information together and it is all easily accessible throughout the game play.

1. **Saving game moves**

An array is not ideal for storing user’s moves because it is not dynamic therefore a doubly linked list was needed to be able to add an unspecified amount of user’s moves and to easily find the latest move and delete a move for the assisted mode. A separate source file with Doubly Linked List struct and its methods was created and then made into a library to be linked with the main game source file to make the main file less confined.

In the normal gameplay, Doubly Linked List acts as a dynamic array that stores all moves to allow the game to be replayed later. In the assisted mode that allows a player to undo and redo their moves, it saves both the gameplay moves and all undone moves. When the player wants to undo their move, a method starts at the end of the doubly linked list and makes its way back to find the latest move made by the player. The undone move is then saved in the undoneMoves doubly linked list. When a user wants to redo their move, the same algorithm is used but on the undoneMoves list to find the latest undone move made by the player.

When the player wants to re-watch a game, the linked list acts as a queue, it uses its first elements to recreate players moves.

1. **Saving games history**

To display unspecified number of past games and then replay them, another list was needed. A LinkedList structure was used to create a GameHistory struct that stores a list of moves taken in a game(a pointer to a Doubly Linked List), two players (pointers to Player structs) and a pointer to the next GameHistory struct. There was no need to implement the previous struct link, therefore a Singly Linked List seemed like the best option.

## Algorithm to check for winners

After every turn, the program needs to check for possible winners. To achieve that, a linear search algorithm with a counter was deployed to check the char array (board) for a horizontal, vertical, or diagonal connection of four of a kind.

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.

**Table 1: Check horizontally: moving from the start point to the right**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |

To check it vertically, two loop 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 – minus 7 for the 6x7 board.

**Table 2: Check vertically: move from the start point up**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 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 | | | | | | |

To check for winners diagonally down, two nested loops were needed: one to check the first half of the board starting from the beginning of each row and moving diagonally down by adding an offset of 8 for the 6x7 board, and a second one to check the second half of the board starting at the top of each column and moving diagonally down again by adding an offset of 8.

**Table 3: Check diagonally down: moving from the start point and going diagonally down.**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 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 row, the loop moves up by subtracting 6 for 6x7 board and for the second half of the board, it starts at the bottom each column moving up by subtracting 6.

**Table 4: Check diagonally up: moving from the start point and going diagonally up**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 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 | | | | | | |

## Linked List and Doubly Linked List algorithms

# Critical Evaluation

Features that work well: single player and multiplayer, saving user’s moves and game history.

Things that could be better:

Single player mode could have difficulty settings. For now, the “computer” only uses randomly generated moves.

The board size is set. It would be good to make it adjustable.

# References

**Wikipedia 2021, *Connect Four*, viewed on 22 March 2021,** **https://en.wikipedia.org/wiki/Connect\_Four**