Algorithms and Data Structures Coursework 1 Report

Jonathan Binns

40311703

Introduction

The main objective of the task is to create a functioning noughts and crosses game, while displaying our skills in implementing different data structures and algorithms to act on those data structures. In order to create a noughts and crosses I need to represent the different aspects of the game. These are the game board, the players, the pieces and the positions the pieces are played at. This can be done with different data structures and how I will represent these will be discussed in the design section. There will also be some extra features implemented in this game. The first is the ability to display instructions so the user knows how to play the game. The second feature is adding the ability to undo, and redo moves made so players can fix any mistakes they make. Finally, the last feature will be the ability to record games, so they can be replayed at a future time.

Design

My design approach centers around what needs to be represented in this game and how I could go about representing them. The first, and most important, thing I need to represent is the game board itself. I will use a one-dimensional array of nine spaces to do this. The main reason for this is that it's the simplest approach which means I can build my functions around it easily as I can calculate which positions in the array equate to each position in a board. It also helps to minimize the number of loops I will need when dealing with it, as there is only one 'row' I only need one loop to go through each element. This also means that my stack can be a two-dimensional array, which is easier to implement than a three dimensional array. The next thing I need to represent is each piece played by the users. For this I will be using a char, either an 'X' or an 'O'. This means I can make the array that represents the board an array of chars. My reason for representing moves like this is that a noughts and crosses game is typically played by writing an 'X' or an 'O' so by representing each move with those characters it allows the game to look as natural as possible. Each piece also needs a place on the board to go, to make sure they go in the intended place. I will represent the board positions as integers. These integers will correspond to a position in the array meaning that an insert function can take in both a char and an int to represent the move made and the position on the board it is made to. This allows them to be inserted quickly as there only has to be one loop to add a move to the board array. My undo and redo move features will be represented by a stack. My plan is to have a stack of boards and add a board after a move has been made. This means that when calling the undo or redo feature the top board on the stack must be made equal to the board that is currently in play. This means that there is no need to represent moves as individual structs and it takes out any errors in terms of people wanting to undo a move that changes a space on the board that is not null. Finally, to allow the user to replay games at a future date I will add the stack of all boards to a .txt document at the end of each game. This is the best way to represent this as I already have to keep a stack for undoing moves so printing this to a .txt document is not a hard extension of that.

Enhancements

The first thing I would improve in my program is having my main menu become a function rather than a set of print statements in the main method. To do this I would pass my mainMenuChoice integer through as a parameter that gets changed based on the user input in the command line.

Then this would correspond to the if-else statements in the main method. I would also do this with my game menu and pass the gameMenuChoice integer as a parameter that would get changed to decide what the next thing to do in the game is. A second thing I could add to my program is give it the ability to scale the board up to a 4 by 4 or maybe a 5 by 5 grid. To do this however, I would probably have to re-write most of my code so that the board was a two-dimensional array rather than a one-dimensional array. I would do this because I could re-write functions like checkWinner() and display() to use loops rather than hard coding them to work with a 9 space array. This means that I could use all my functions for any sized board, keeping the repeated code to a minimum and keeping the main method concise. Another feature I could add is an 'A.I.' for the user to play against. To make this I could start by having an algorithm to loop through the board and randomly pick a free space to play a move. This is the most basic form of an A.I. I could think of. I could continue to make it more sophisticated by using if-else statements to tell it to block any rows or columns the user is close to completing. A final improvement to the A.I. could be choosing a strategy out of a few possible ones to try and win the game. The strategy would just be a set of moves to be played however that would allow the A.I. to play aggressively instead of always trying to play defensively, making it more challenging for the user to play against. The final feature I would add if I had more time is a 'heat map' which would show the number of people who played in each square after the first move has been played. This could be implemented by having each starting move and second move recorded in one txt file and each time someone plays a second move a counter would increase on the positions that were played. It would be displayed by showing a number in each position in the board once the first move has been played. This could be helped to improve my A.I. by having it always choose the most popular move that has been played after each move the user makes. However, this would take a lot of time, and would continually slow down the game as the A.I. would get more and more complicated with more if-else statements for it to get caught up in.

Critical evaluation

Overall, I think my functions work well, I achieved this by making sure they were all as simple as possible to keep processing time down. I also did this by, where possible deciding to hard code all the possible combinations for my array than using a loop, which then could be scaled. My display function was one of the first I wrote, and I think it works well. This is an example of me 'hard coding' the function rather than using a loop. This function takes in an array of characters. The first three are displayed on the first line, the second three on the second line and the last three on the bottom line. To make it look like a board I added characters such as | and – to separate each square (Geeks for Geeks. 2019). The second function I think works well is my addMove() function. This function takes in an array of characters, asks the user for a position and a move to be played. Then it checks that the position in the board that they have chosen is null. After that it calls my insert function which takes in a char array, an int and a char. The array for the board, the int for the position and the char for the move. My insert function just inserts the move played at the position specified in the array. It also checks that the character is a capital X or a capital O. After adding the move the user wants to make to the array addMove() then calls my display function. I think this function works well as it does everything needed to safely add a move within a small number of lines. It also works very fast with little or no input lag or time taken after the user types in a move. My check winner function is one of the functions I could improve given more time. This function takes in, a char array, for the board, an int, for the winner variable which becomes 1 when a winner is found, and finally a char for the winning player. The function uses a series of if-else statements to go through all the possible combinations of a win by looking at which positions in the array are equal to which character. If a winner is found the winner integer is set to one, breaking the loop in the main method, and the winning player character is set to which character has matched the positions in the array. If I had more time I could improve this by using a set of loops to check for the rows, and columns and only two if else statements for diagonal wins. This would also mean id have to re-write my array as a two dimensional one as looping through my one-dimensional array would take roughly the same amount of code and would take a little longer to execute. The next function I think works well is my check full board function. This function takes in a char array and an integer, the array represents the board and the int represents a variable that

becomes 1 when a full board is detected. The function loops through the board and for each position that is not null it increments a counter. Once the loop has gone through all elements in the array there is an if-else statement that checks if the counter is 9, in which case there is a full board. Otherwise the full board variable is kept at zero. The full board variable is used in my main method to stop asking the user to input moves once they have filled the board. I think this function works well as the loop will at a worst case go through 9 elements, this keeps it quick to run as 9 elements is not a lot. Alongside that at a worst case it is incrementing the variable 9 times but at maximum it must do this once a game, as the board is full. I think My stack functions work well overall. The first function I wrote was the initialize function. This takes in the stack and sets every element to be null. This taught me a lot about a stack as a concept and two-dimensional arrays. The next function I wrote was the display function, this takes in a stack and loops through each 'layer' and calls my display function to display the elements in that array. This was mostly used for testing, to make sure that my initialize function worked and to test my push function before I started work on my pop function. This function is quick as it only have to loop through 9 elements, and because my display function doesn't include any loops each array can be printed as quickly as possible. The next stack function I wrote is the push function. This takes in a stack and a char array. It then increments the value for the top of the stack and loops through the char array, representing the board, and sets it equal to the current layer in the stack. This is also very efficient as it only must do one loop of nine rather than one for the layer and one for each element in that layer. Finally, I wrote my pop function, this takes in a stack and a char array. This function is slightly different to a normal pop function as instead of returning the value of the top of the stack it replaces the game board with that array. The function first decrements the value for the top of the stack. It then loops through the elements in that layer of the stack and sets them equal to the elements of the board in play. It then calls my display function to show the board. Overall, all my stack functions are concise and efficient with the biggest loop only needing to be done once a game. Aside from that I make sure to keep loops to a minimum and I call as many other functions as I can to keep the main method as quick as possible. The final Stack function is the redo move function. If I had more time this could be improved by integrating it into the main stack instead of having a second stack to deal with re-doing moves. Currently the redo feature can only redo a move once before glitching and causing errors. With more time these errors could be corrected and the redo feature would work just as well as the undo feature. My final two functions are my read and write file functions. The write file function takes in a stack, it then creates a pointer to a file and gets the time, in seconds since the first of January 1970. The function then creates a file called game(time).txt and opens it in write mode. The function then moves through each layer of the stack and writes them to the txt file in the same format that they are printed. I couldn't use my display function here as that uses printf whereas I needed to use fprinf. This doesn't affect efficiency but if I had more time I would have liked to try and have a function to actually write the arrays to the file rather than using near identical code twice. After the function has looped through all layers of the stack it closes the file. Next my read function this doesn't take in any parameters. It first creates a pointer to a file, it then asks the user to enter the name of the txt file that holds the game they want to replay. It then opens the file in read mode, and after checking that the pointer isn't null, loops through all the characters in the txt file and prints them to the screen. After that it closes the file. The only problem with these functions is that sometimes the write function can hang for a second or two when generating the time. Asides from that both are efficient and do their job quickly.

Personal Evaluation

The first thing I learned to complete this coursework is how arrays work. I already had a basic understanding of this but for this coursework I needed to learn ways to display and add to them in ways that I never had before. An example of this is my insert function, it inserts a value at a position in an array, this taught me how to add to an array in a concise way by calling a function rather than creating a loop to add to an array each time I needed to do so.

Stacks/structs

The second type of array I needed to learn is a two dimensional array, I used this in my stack. This took some time to get my head round as I had never used one in this way before. For my stack I needed to use the two dimensional array to store my list of boards and to add and remove them from the list I had to re-think how I worked with arrays. Making sure to keep the column consistent while I looped through each element of the row. The lab also helped me to learn about stacks as well, the push, initialize and pop functions were all easy to understand once I had got my head around how id implement them in a two dimensional array rather than a one dimensional array. The last part of my stack I had to learn was a struct, I had used this datatype before but not to create a stack. However, using it as a stack helped me to use it just like any other data type, passing it around as a parameter into other functions. The final thing I improved is my skills with functions. This coursework helped me to learn a lot about them and features such as passing parameters and changing values that are declared in the main within a function. The passing of parameters through a function was the first thing I needed to learn as the easiest place to start this coursework was having a display function working and to get this working I needed to pass an array to be printed. The next thing I needed a function to do was change a value that is declared in the main. This was done in the check winner function as the winner integer is the condition of my main while loop in the main function, and in order to break that loop I needed to change the value of the integer from zero to one.

The biggest challenge I faced was implementing the stack. I needed to learn both how my struct worked and how that related to my two dimensional array contained within it. After studying the stack implementation in the lab I eventually realized that the top integer declared in my struct should be used to keep track of the row in my array, this would be the number of each 'layer' of the stack. This helped me to implement both the push and pop functions as I realized that all I needed to do was loop through the columns of each row and only increment or decrement the top value once per function. The next challenge I faced was making sure in the main method the user was only asked for moves when it was valid, i.e. there is no winner and the board isn't full. To overcome this, I used two variables to make sure that if there was a winner or the board became full the while loop would break and the user wouldn't be asked for any more moves. These variables get passed to functions which check for both of these events, checkWinner() and checkFullBoard() and when either one is true the while loop breaks and a set of if-else statements declare what will happen afterwards.

Overall, I think I performed well as I reached all of the minimum requirements set out in the description and I went above them when I included my undo feature and my replaying an old game feature. Alongside this my program runs quickly and efficiently and is easy to understand and use for a new user.

References

Geeks for Geeks. (2019). Implementation of a tic-tac-toe game. Accessed On: 25/03/19. Used For: Function ideas, display function spacing and formatting
Retrieved From: https://www.geeksforgeeks.org/implementation-of-tic-tac-toe-game/

Stack Overflow. (2015). Updating an int inside of a function. Accessed on 25/03/19. Used For: updating int variables inside functions, e.g. winner in checkWinner().
Retrieved From: https://stackoverflow.com/questions/23667497/update-int-variable-in-c-inside-a-function

CodeGuru. (2004). Adding an int to a string. Accessed on 25/03/19. Used For: creating the .txt title in writeFile().
Retrieved From: http://forums.codeguru.com/showthread.php?405484-how-do-i-concatenate-a-string-with-an-integer

Simon Wells. (2019). Lab 3. Accessed on 25/03/19. Used For: array insert function, array initialize function, stack struct, stack push, stack init and elements were taken for stack pop;