Chess program report

• Description of the application and features:

The application provides a chess game experience human versus human with the ability to undo or redo moves for both players, saving the game and loading it at any time during the game and at the correspondent player's turn .The app also displays the taken out pieces for both players and the list is modified each turn according to the moves. Any input is checked and a proper message is displayed in case of invalid moves. Some features are included such as check, checkmate, promotion and stalemate and the player is notified in each case with a message. The game can end in a draw by stalemate or won by checkmate.

• Overview of the design ,description of used data structures and of the important functions/modules in the implementation.

The code is divided into 3 main parts:

I. Before main function: (global variables and functions prototypes)

I. The global variables used :

- `char Board[10][10];`

  the main 2D array of size 10x10 representing the board filled with pieces where the whole game takes place from beginning to end through the pieces' moves

- `int maxTurn = 0;`

  a counter(integer variable) that increases in each turn and by the end of the game it represents the maximum number of the played moves (that counter helps in the undo and redo steps to limit any further undo or redo moves out of the range)

- `int currentTurn = 0`

  a counter (integer variable) that increases in each turn and decreased for the undo in the undo function or increased in case of the redo

- `int player=1;`

a counter (integer variable) that changes the player's turn used for switching between the 2 players

- int row_1,col_1,row_2,col_2;

  4 integer variables requires as input from the user and indexes of the 2 squares describing the move (from and to) to access any piece in board
- char promotion;

  a character variable used to get the desired promotion of the pawn from the user when the pawn reaches the board limits
- char EmptyBoard[10][10];

  a 2D array 10x10 representing an empty chess board used in moving pieces from one square to another and assigning the left square with the corresponding one of the empty board
- char Alternative[10][10];

  a 2D array representing an alternative board with the current positions of pieces in a specified turn to examine any possible moves for the king as a way-out from an imminent checkmate or stalemate (used for check ,stalemate and checkmate)
- char piecesout1[16];

  an array 1D of size 16 contains the list of the taken out pieces for player 1
- char piecesout2[16];

  an array 1D of size 16 contains the list of the taken out pieces for player 2
- char input[4];

  an array of characters 1D sized 4 to get input (the indexes of the 2 squares where the piece is moving from and to) from the user
- struct turn

```
{
   char board[10][10];
   char piecesout1[16];
   char piecesout2[16];
   int k;// counter to increase the counter of the array of the taken out pieces for player1
   int m; // to increase the counter of the array of the taken out pieces for player2
};
struct turn turns[1000];//TO SAVE MOVES
```

/*the 1D array of structures of size 1000 saves moves by increasing the counter "currentTurn" for the undo, redo, save and load steps and hence the maximum number of played games to be saved is 1000 games */

II. The principal functions’ prototypes of the game:

- void copyPiece(char pieces[16], char pieces2[16]);

a function that copies the taken out pieces from an array to another used for the undo, redo, save, and load steps

- void copyBoard(char board[10][10], char board2[10][10]);

a function that copies the current board with the current positions of pieces from an array to another used for the undo, redo, save, and load steps

- void initialize_Board();

  a function that arranges the pieces in the board (first view) before any played game

- void displayPlatform();

  a function that prints the array and displays the modified chess board in each turn as well as the modified list of the taken out pieces for both players in each turn

- int check_col_1(char x);

  to validate user’s first input (the index of the column of the current position of the piece)

- int check_row_1(char x);

  to validate user’s second input (the index of the row of the current position of the piece)

- int check_col_2(char x);

  to validate user’s third input (the index of the column of the new position of the piece)

- int check_row_2(char x);

  to validate user’s fourth input (the index of the row of the next position of the piece)

- void moveValidation();

  to validate the next position of the piece required to be moved and that according to each piece allowed movements for each player in the board

- void saveGame(void);

  to save game at any time during the game

- void loadGame(void);

  to load the saved game at any time during the game

- void undo();

  to undo a move

- void redo();

to redo a move

- int Apply_move(int row_1,int col_1,int row_2,int col_2);

to move the required piece from one place to another after the move had been checked and validated

- int King1_row();
- int King1_column();
- int King2_row();
- int King2_column();

to locate the current position of the king for the check, checkmate and stalemate functions for both players

- int CheckKing_1(int x,int y);

  to determine if the player 1’s king is under a check where x and y are the indexes of the row and column of the position of the king

- int CheckKing_2(int x,int y);

  to determine if the player 2’s king is under a check where x and y are the indexes of the row and column of the position of the king

- int Check_mate_king1();

  to determine if it is a checkmate case for player 1 or not

- int Check_mate_king2();

to determine if it is a checkmate case for player 2 or not

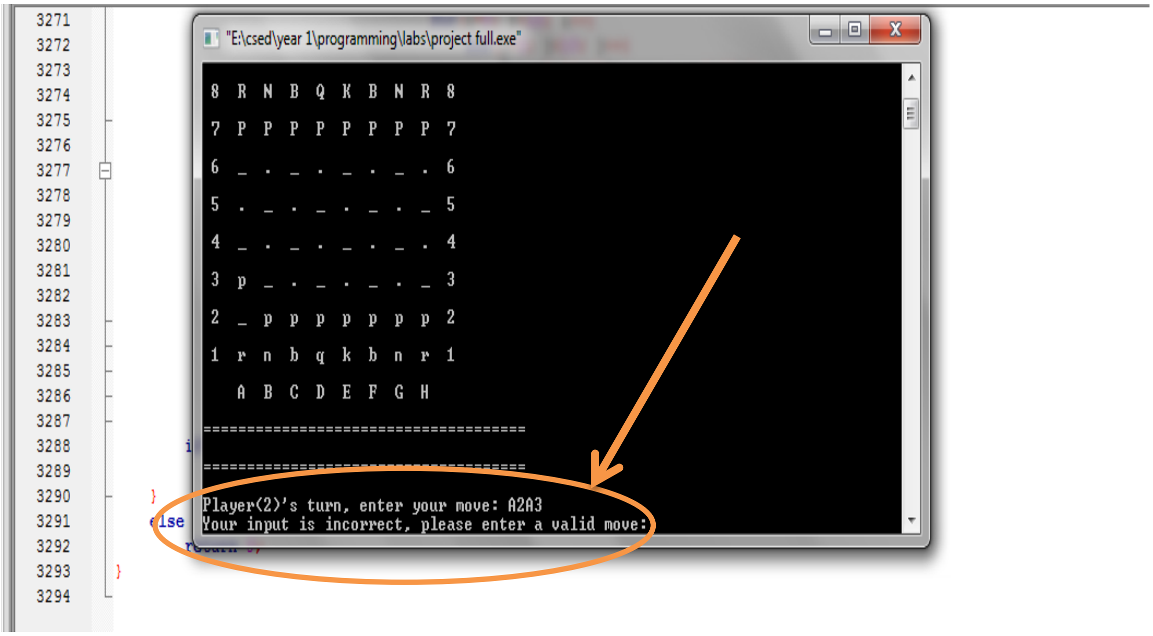
II. Main function:

The main function contains the main flow of the game :

1. Through function calls for the above functions, the empty board is defined by filling an array of size 10x10 with ”.” and ”_” consecutively describing black and white squares of the chess board while keeping the border rows and columns for the traditional indexes of the chess board then filling 8x8 of the same array with the corresponding pieces
2. Then the whole game takes place in a big while loop:

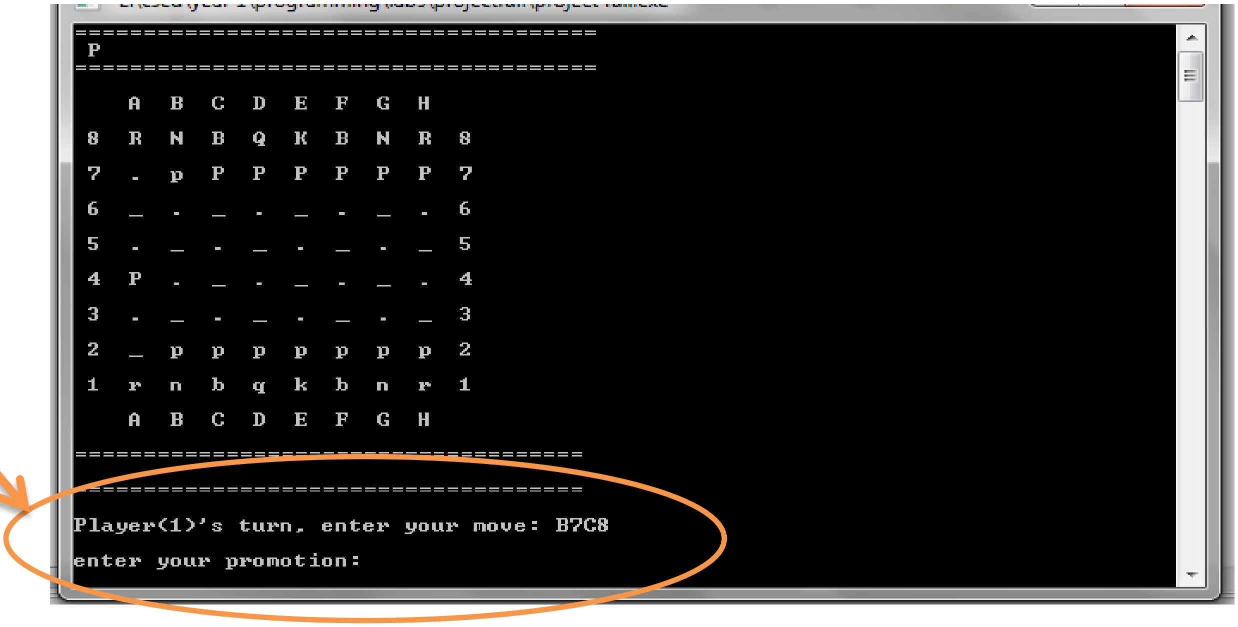
For the first player: an input is asked from the user and through “moveValidation() “function ,the input is accepted or not .In
case of rejection ,the user is asked for another one. (and in case of promotion: an input is asked from the user)

Here is a sample run of the move validation: A2A3,A2A3



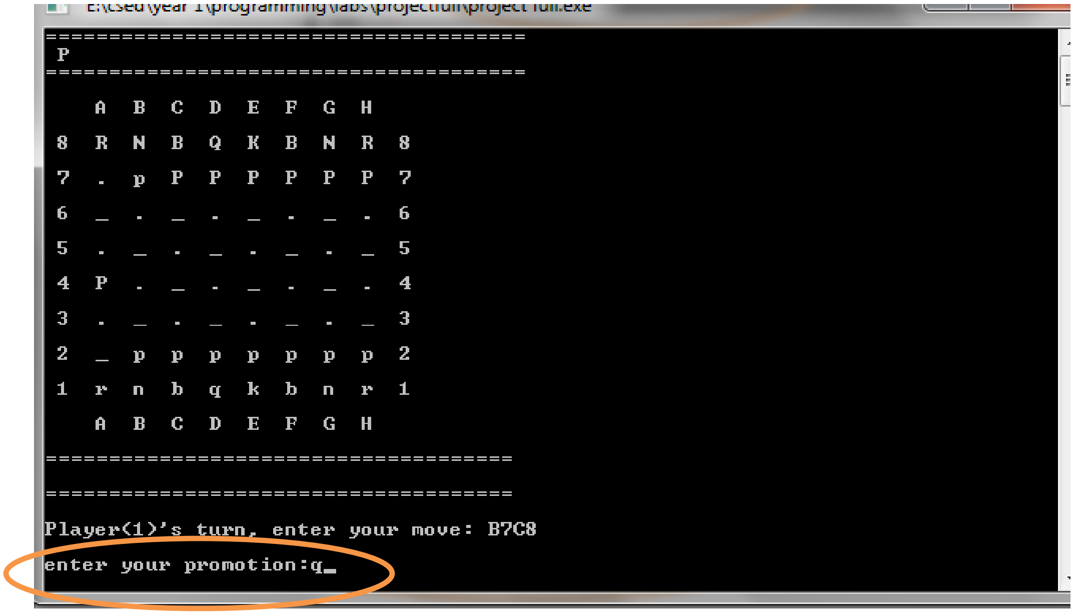
```
8 R N B Q K B N R 8
7 P P P P P P P P 7
6 _ . _ . _ . _ . 6
5 . _ . _ . _ . _ 5
4 _ . _ . _ . _ . 4
3 p _ . _ . _ . _ 3
2 _ p p p p p p p 2
1 r n b q k b n r 1
  A B C D E F G H
=====================================
=====================================
Player(2)'s turn, enter your move: A2A3
Your input is incorrect, please enter a valid move:
```

Case of promotion:

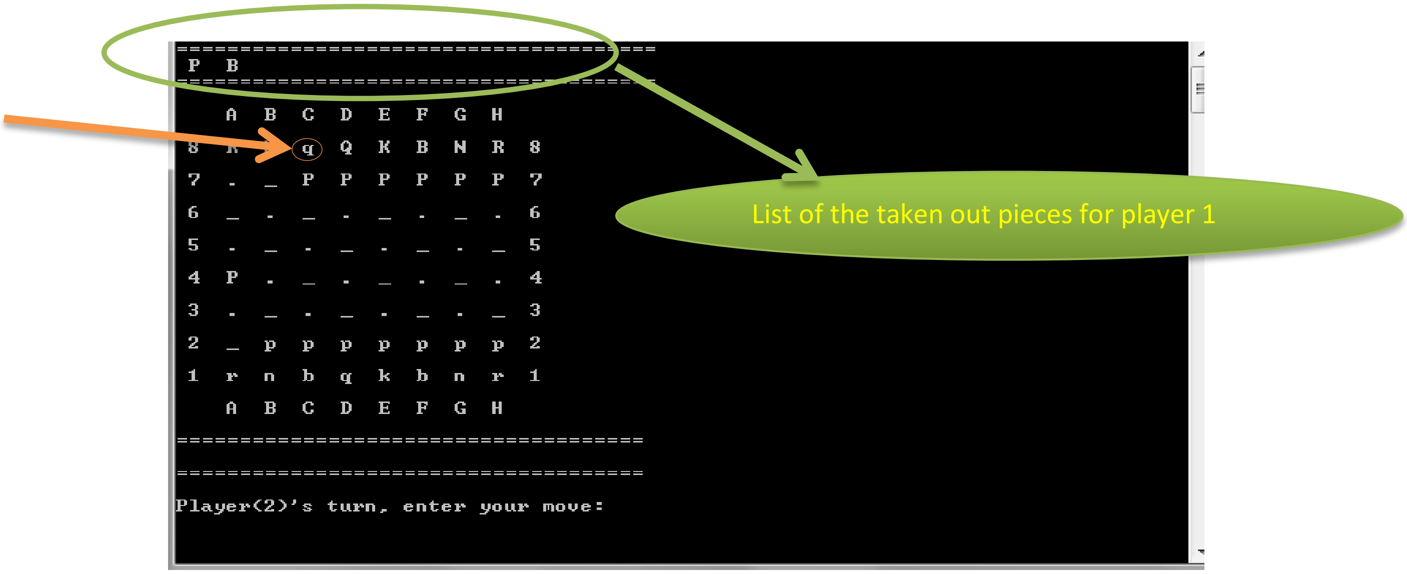


```
======================================
 P
======================================

    A   B   C   D   E   F   G   H
8   R   N   B   Q   K   B   N   R   8
7   .   p   P   P   P   P   P   P   7
6   _   .   _   .   _   .   _   .   6
5   .   _   .   _   .   _   .   _   5
4   P   .   _   .   _   .   _   .   4
3   .   _   .   _   .   _   .   _   3
2   _   p   p   p   p   p   p   p   2
1   r   n   b   q   k   b   n   r   1
    A   B   C   D   E   F   G   H
=====================================
=====================================
Player(1)'s turn, enter your move: B7C8
enter your promotion:
```

Promotion of the first player pawn into a queen



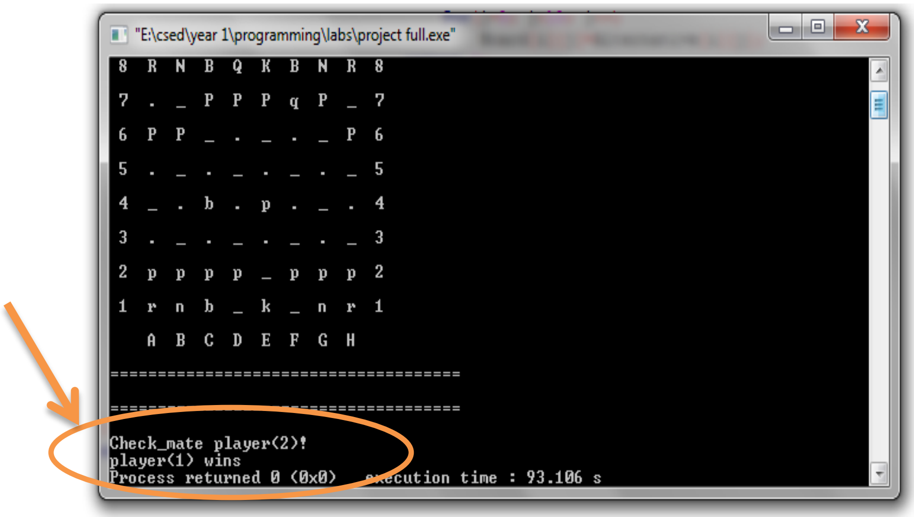
```
======================================
 P
======================================

    A   B   C   D   E   F   G   H

 8  R   N   B   Q   K   B   N   R   8

 7  .   p   P   P   P   P   P   P   7

 6  _   .   _   .   _   .   _   .   6

 5  .   _   .   _   .   _   .   _   5

 4  P   .   _   .   _   .   _   .   4

 3  .   _   .   _   .   _   .   _   3

 2  _   p   p   p   p   p   p   p   2

 1  r   n   b   q   k   b   n   r   1

    A   B   C   D   E   F   G   H

=====================================
=====================================
Player(1)'s turn, enter your move: B7C8
enter your promotion:q_
```



```
======================================
 P  B
======================================

    A   B   C   D   E   F   G   H

 8  R   q   Q   K   B   N   R   8

 7  .   _   P   P   P   P   P   P   7

 6  _   .   _   .   _   .   _   .   6

 5  .   _   .   _   .   _   .   _   5

 4  P   .   _   .   _   .   _   .   4

 3  .   _   .   _   .   _   .   _   3

 2  _   p   p   p   p   p   p   p   2

 1  r   n   b   q   k   b   n   r   1

    A   B   C   D   E   F   G   H

=====================================
=====================================
Player(2)'s turn, enter your move:
```

List of the taken out pieces for player 1

3. After the move is being validated, the input is tested by the check function ,if it is a check case ,another test of checkmate function is done ,if it is also a checkmate case :the game is won by checkmate and the loop is broken
Sample run: White first: E2E4, H7H6, D1H5, A7A6, F1C4, B7B6, H5F7 (Pawn is eaten and checkmate)

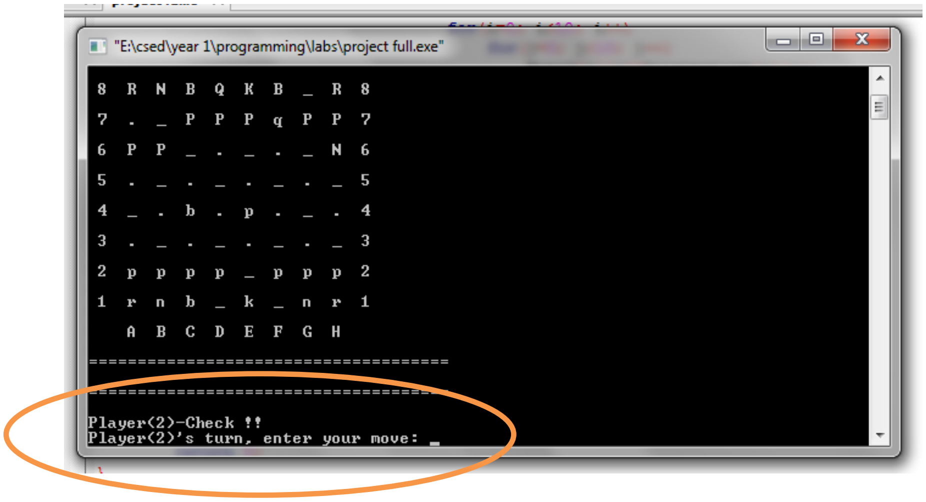


"E:\csed\year 1\programming\labs\project full.exe"

```
8  R  N  B  Q  K  B  N  R  8
7  .  _  P  P  P  q  P  _  7
6  P  P  _  .  _  .  _  P  6
5  .  _  .  _  .  _  .  _  5
4  _  .  b  .  p  .  _  .  4
3  .  _  .  _  .  _  .  _  3
2  p  p  p  p  _  p  p  p  2
1  r  n  b  _  k  _  n  r  1
   A  B  C  D  E  F  G  H
=====================================

=====================================

Check_mate player(2)!
player(1) wins
Process returned 0 (0x0)   execution time : 93.106 s
```

4. But if it is check and not checkmate , a proper message is displayed to warn the player who continues to play normally and his next move is again tested

   Sample run: Whitefirst:E2E4,G8H6,D1H5, A7A6,F1C4,B7B6,H5F7(Check),

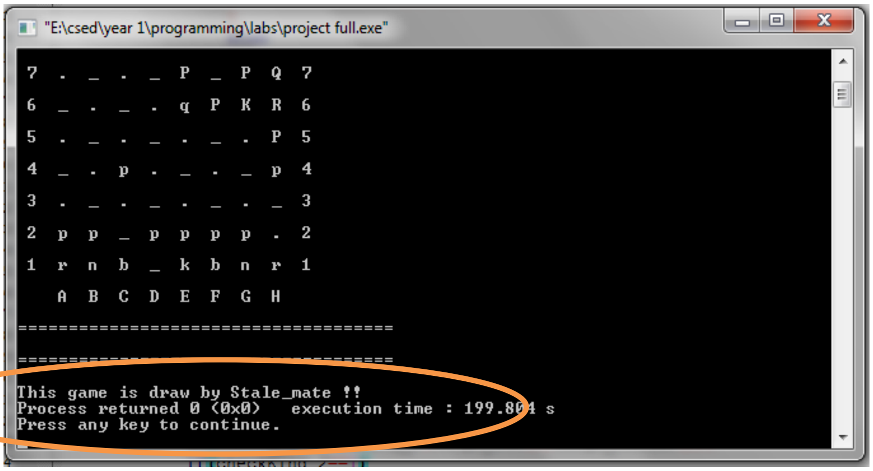


"E:\csed\year 1\programming\labs\project full.exe"

```
8  R  N  B  Q  K  B  _  R  8
7  .  _  P  P  P  q  P  P  7
6  P  P  _  .  _  .  _  N  6
5  .  _  .  _  .  _  .  _  5
4  _  .  b  .  p  .  _  .  4
3  .  _  .  _  .  _  .  _  3
2  p  p  p  p  _  p  p  p  2
1  r  n  b  _  k  _  n  r  1
   A  B  C  D  E  F  G  H
=====================================

=====================================

Player(2)-Check !!
Player(2)'s turn, enter your move: _
```

5. In case of not check but checkmate, then the game is draw by stalemate

   Sample run: White first : C2C4, H7H5, H2H4, A7A5, D1A4, A8A6, A4A5, A6H6, A5C7, F7F6,C7D7 (Check),E8F7, D7B7 ,D8D3 ,B7B8 ,D3H7 , B8C8,F7G6,C8E6



```
7  .  _  .  _  P  _  P  Q  7
6  _  .  _  .  q  P  K  R  6
5  .  _  .  _  .  _  .  P  5
4  _  .  p  .  _  .  _  p  4
3  .  _  .  _  .  _  .  _  3
2  p  p  _  p  p  p  p  .  2
1  r  n  b  _  k  b  n  r  1
   A  B  C  D  E  F  G  H
====================================

====================================
This game is draw by Stale_mate !!
Process returned 0 (0x0)   execution time : 199.804 s
Press any key to continue.
```

6. If it is not any of the above cases the steps 2,3,4,5 are checked for the other player
7. then the player is switched and an input is asked from the user
8. then the game is looped from step 1 to 7 till any of the (2 to 5 conditions stated above) breaks the loop

III. After main function:

Are the detailed descriptions for the functions ‘prototypes stated above

Implementation assumptions made and details of the principal functions:

1. For the undo,redo,save,load part:

(a) An array of structures 1D of size 1000 is used .Each element of the array has 5 members:

```
• Struct turn{
  char board[10][10];
  char piecesout1[16];
  char piecesout2[16];
   int k;// counter to increase the counter of the array of the taken out pieces
   for  player1
    int m; // to increase the counter of the array of the taken out pieces for player2
     };
struct turn turns[1000];
```

***in each turn the move is saved completely saved: with the view of the board , the list of the taken out pieces for both player ,the counters of these lists are also increasing in each turn .as well as the counter of the array of structures .By the end of the game this counter reaches its maximum. and hence:

for the undo: if the current counter in the current play is>0 the undo can de done, else a message is displayed saying :you can't undo any further

for the redo: if the current counter < maximum counter-1 the redo can be done ,else a message is displayed saying :you can't redo any further

- int maxTurn = 0;

  a counter(integer variable) that increases in each turn and by the end of the game it represents the maximum number of the played moves (that counter helps in the undo and redo steps to limit any further undo or redo moves out of the range)
- int currentTurn = 0

  a counter that increases in each turn and decreased for the undo in the undo function or increased in case of the redo

(b)void copyPiece(char pieces[16], char pieces2[16]);

a function that copy the taken out pieces from an array (list of the taken out pieces)to another (an alternative one) to return them again to the main list in case of the undo, redo, save, orload

(c)void copyBoard(char board[10][10], char board2[10][10]);

a function that copy the current board with the current positions of pieces from an array (the main board)to another (an alternative one) to return them aagain to the main board in case of undo, redo, save, or load

*** *saving the game can be done any time during the game as well as loading where the game will be loaded with the corresponding player's turn*

2. System ("cls");

That function clears the screen in each move so that one and only board is displayed on screen , this function is repeated before each displayPlatform();function which prints the array.

3. For the check,checkmate and stalemate part:

- **int King1_row();**

  -locating the row of player 1 's king

-if the king is found ,the flag value “king” is assigned to 1 and the function returns the index of the row-1.
-if not found, the flag “king=0”

- **int King1_column();**

  -locating the column of player 1 ‘s king
  -if the king is found ,the flag value “king” is assigned to 1 and the function returns the index of the column-1.
  -if not found, the flag “king=0”

- **int King2_row();**

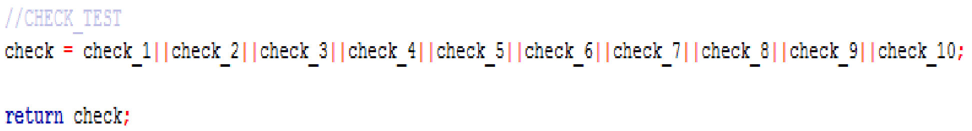
  -locating the row of player 2 ‘s king
  -if the king is found ,the flag value “king” is assigned to 1 and the function returns the index of the row-1.
  -if not found, the flag “king=0”

- **int King2_column();**

  -locating the column of player 2 ‘s king
  -if the king is found ,the flag value “king” is assigned to 1 and the function returns the index of the column-1.
  -if not found, the flag “king=0”

- **int CheckKing_1(int x,int y);**

  -to determine if the player 1’s king is under a check where x and y are the indexes of the row and column of the position of the king
  -this function takes the return values of the 2 functions int King1_row(); and int King1_column(); and check the presence of any of the other pieces in the board using for loop for each row of same index then return the flag value “check1=1”in case of check
  -the latter step is then repeated for each row and each column having same indexing in case of presence of other pieces and at each time returning a new value for a new flag(check2,check3,…. and so on…)
  -then



```
//CHECK_TEST
check = check_1||check_2||check_3||check_4||check_5||check_6||check_7||check_8||check_9||check_10;

return check;
```

So if “check=1”,there is a check on the king1

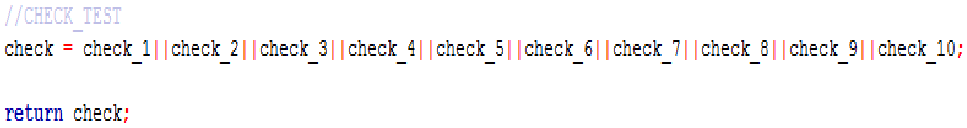
- **int CheckKing_2(int x,int y);**

-to determine if the player 2’s king is under a check where x and y are the indexes of the row and column of the position of the king

-this function takes the return values of the 2 functions int King2_row(); and int King2_column(); and check the presence of any of the other pieces in the board using for loop for each row of same index then return the flag value “check1=1”in case of check

-the latter step is then repeated for each row and each column having same indexing in case of presence of other pieces and at each time returning a new value for a new flag(check2,check3,…. and so on…)

-then



```
//CHECK_TEST
check = check_1||check_2||check_3||check_4||check_5||check_6||check_7||check_8||check_9||check_10;

return check;
```

So if “check=1”, there is a check on the king2

- **int Check_mate_king1();**

-to determine if it is a checkmate case for player 1 or not

-this function takes also the return values of the 2 functions int King1_row(); and int King1_column(); and check the presence of any of the other pieces in the

-then using a for loop to check the presence of other pieces, the checkmate function also uses int CheckKing_1(int x,int y);to test rows or columns of the same indexing and the return value is assigned to different flags of prefixes ”check” in case of check and ”out” in case of not check for each row or column

-then these return values are grouped as follow :

“

int check_around = (check_1||out_1)&&(check_2||out_2)&&(check_3||out_3)&&(check_4||out_4)&&(check_5||out_5)&&(check_6||out_6)&&(check_7||out_7)&&(check_8||out_8);

”

-if *check_around ==1,*then

1. 2 For loops on the entire board to Copy the entire board with the current positions of pieces to an alternative one char Alternative[10][10]; to be able to apply an imaginary move and see if it will cause a checkmate or not on the king
2. 2 for other loops inside the above two to check the presence of any other pieces other than the king in the board and when found:

3. 2 for loops inside the above two loops where the indexes of the square in each case are copied to a temporary variable
4. Then testing a move of the piece found between the indexes found in step 1 and step 2(i1,j1,i2,j2) to see if it is valid
5. After the move is tested and approved by the correspondent function, The king is again exposed to a check test:
6. If not check , there is no danger ,copy the alternative board to the main one and the play is resumed
7. If check the move is advanced meaning

```
                    Board[i1][j1]=Board[i2][j2];
                     Board[i2][j2]=temp;
                     for(i=0; i<10; i++)
                       for(j=0; j<10; j++)
                      Board[i][j]=Alternative[i][j];
```

8. And the loop will again test the new position and so on till declaring checkmate or not

- if *check_around ==0,* the game is resumed according to the conditions of the main function and no checkmate

- **int Check_mate_king2();**

-to determine if it is a checkmate case for player 2 or not

-this function takes also the return values of the 2 functions int King2_row(); and int King2_column(); and check the presence of any of the other pieces in the

-then using a for loop to check the presence of other pieces, the checkmate function also uses int CheckKing_2(int x,int y);to test rows or columns of the same indexing and the return value is assigned to different flags of prefixes "check" in case of check and "out" in case of not check for each row or column

-then these return values are grouped as follow :

"

int check_around = (check_1||out_1)&&(check_2||out_2)&&(check_3||out_3)&&(check_4||out_4)&&(check_5||out_5)&&(check_6||out_6)&&(check_7||out_7)&&(check_8||out_8);

"

-if *check_around ==1,*then

1. 2 For loops on the entire board to Copy the entire board with the current positions of pieces to an alternative one char Alternative[10][10]; to be

able to apply an imaginary move and see if it will cause a checkmate or not on the king

2. 2 for other loops inside the above two to check the presence of any other pieces other than the king in the board and when found:
3. 2 for loops inside the above two loops where the indexes of the square in each case are copied to a temporary variable
4. Then testing a move of the piece found between the indexes found in step 1 and step 2(i1,j1,i2,j2) to see if it is valid
5. After the move is tested and approved by the correspondent function, The king is again exposed to a check test:
6. If not check , there is no danger, copy the alternative board to the main one and the play is resumed
7. If check the move is advanced meaning

```
Board[i1][j1]=Board[i2][j2];
  Board[i2][j2]=temp;
  for(i=0; i<10; i++)
    for(j=0; j<10; j++)
   Board[i][j]=Alternative[i][j];
```

8. And the loop will again test the new position and so on till declaring checkmate or not

- if *check_around ==0,* the game is resumed according to the conditions of the main function and no checkmate

4. Moves :

- **void moveValidation()**

-takes the return values of 4 other functions

```
check_col_1(input[0]);
check_row_1(input[1]);
check_col_2(input[2]);
check_row_2(input[3]);
```

that assure the validation of the user's input

-then this function assures that each player is able to move only the allowed pieces for him

White player: k, q, p, r ,b , n

Black player: K,Q, P ,R, B, N

-also assures that the player can't move a piece from an empty square

- in case of invalid move: a message appears with a sound notifying the user and   the function loops again till a valid move is detected

-in case the user inputs “l” ”o” ”a” ”d”, the function applies loadGame(void);

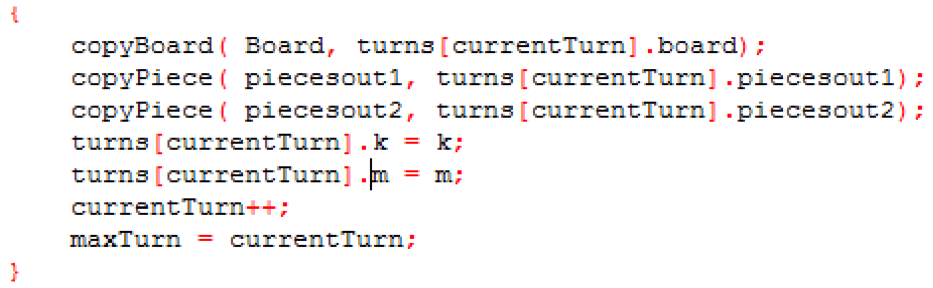
- in case the user inputs “s” ”a” ”v” ”e”, the function applies saveGame(void);

-Also contains these functions system("cls");   and displayPlatform();

- as well as int Apply_move(int row_1,int col_1,int row_2,int col_2) function stated below

- **int Apply_move(int row_1,int col_1,int row_2,int col_2)**

  after the move is being validated by moveValidation(),this function is only responsible to validate the move according to each chess piece allowed movements using for loops and flags
  for example :
  *the pawn is able to move forward only jumping two squares in the first move then only one after that but eats the opponent’s piece from the first square diagonally
  *and for the other pieces according to the chess game rules
  This function also   executes these 5 steps for each input from the user:



```
{
     copyBoard( Board, turns[currentTurn].board);
     copyPiece( piecesout1, turns[currentTurn].piecesout1);
     copyPiece( piecesout2, turns[currentTurn].piecesout2);
     turns[currentTurn].k = k;
     turns[currentTurn].m = m;
     currentTurn++;
     maxTurn = currentTurn;
}
```

  To save moves in the array of structures stated above

- ## User manual:

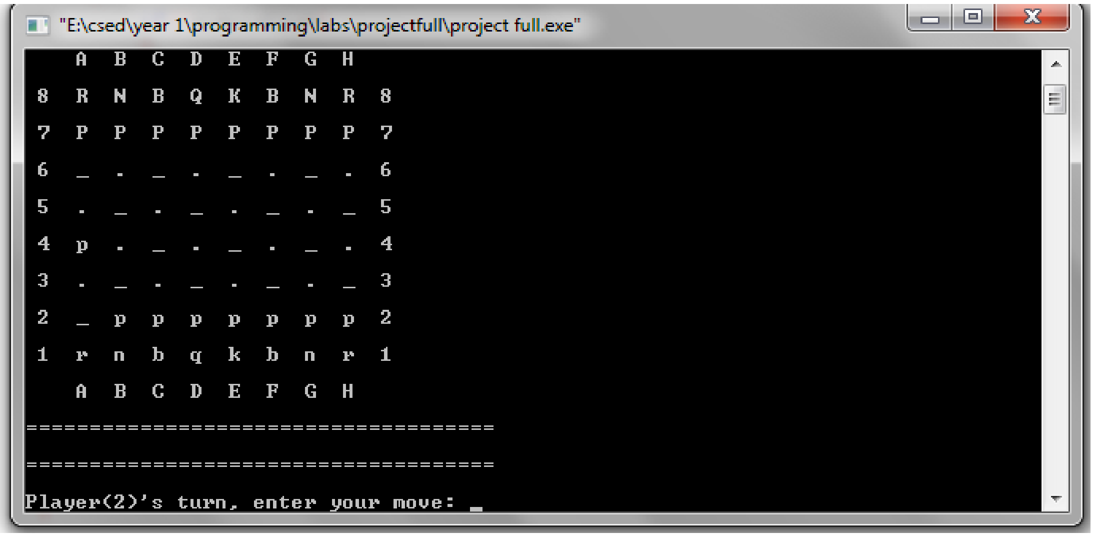
  1. Like any chess game, the first round is for the white player who is only able to move a pawn.
  2. Each square is indexed by a character and a number (e.g. D1,C3,...etc.). The user specifies his next move by entering the index of the piece he wishes to move, followed by the index of the square he wishes to move it to.(e.g. A3B4 will move the piece at A3 to the square B4).

3. When the pawn reaches the board ends , an input is asked from the user to determine the desired promotion for the pawn by entering (q,r,n,b)for the white player and (Q,R,N,B)for the black one
4. If any of the two player is under a check a message is shown notifying him to take correspondent measures
5. If any of the two player wins the game by checkmate, a message appears and no further moves are accepted
6. The game can draw by stalemate when no further moves will be able to save the game and also a message appears in that case to notify both players
7. During the game in case of invalid moves, a message appears with a beep sound and the user is asked to reenter the move
8. The user can save the game at any time during the game by typing “save” instead of the move and the game will be saved then he can then enters the desired move
9. The user can load the saved the game at any time during the game by typing “load” instead of the move and the saved game will be loaded at the correspondent turn of the player
10. The user can undo any move by typing “undo” instead of the move but he can’t undo at the beginning of the game or when he reaches the first move he had entered .if that happens, a message appears: ”you can’t redo any further”

11. The user can redo any move by typing “redo” instead of the move but he can’t redo a move unless he had played so he can’t redo if he reaches the maximum number of played moves .if that happens, a message appears: ”you can’t redo any further”

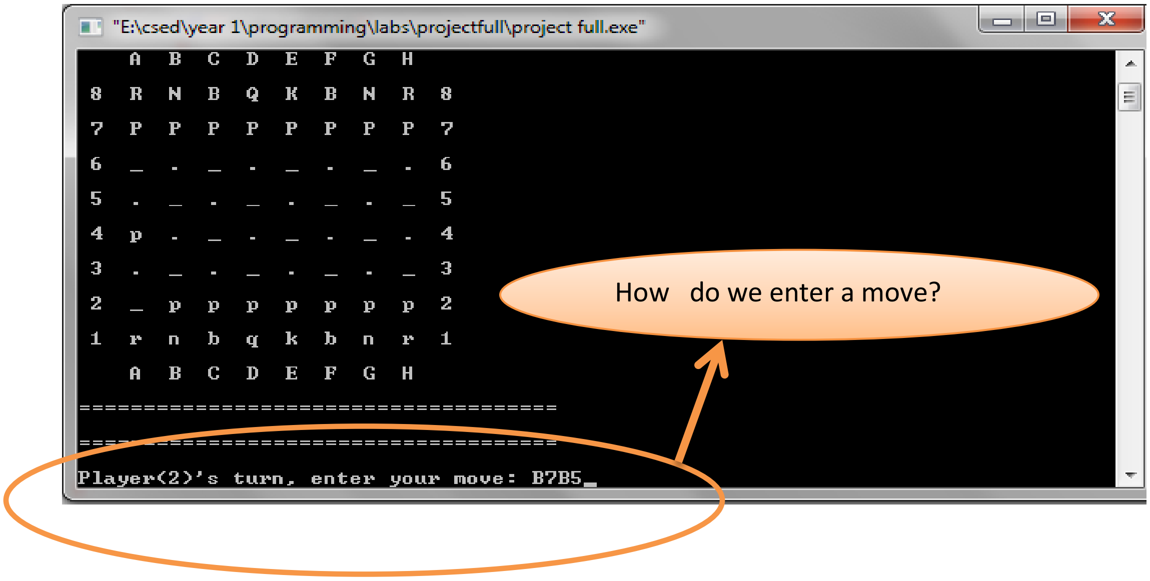
sample runs :

First player :A2A4


"E:\csed\year 1\programming\labs\projectfull\project full.exe"

```
   A  B  C  D  E  F  G  H

8  R  N  B  Q  K  B  N  R  8

7  P  P  P  P  P  P  P  P  7

6  _  .  _  .  _  .  _  .  6

5  .  _  .  _  .  _  .  _  5

4  p  .  _  .  _  .  _  .  4

3  .  _  .  _  .  _  .  _  3

2  _  p  p  p  p  p  p  p  2

1  r  n  b  q  k  b  n  r  1

   A  B  C  D  E  F  G  H

======================================

======================================

Player(2)'s turn, enter your move: _
```

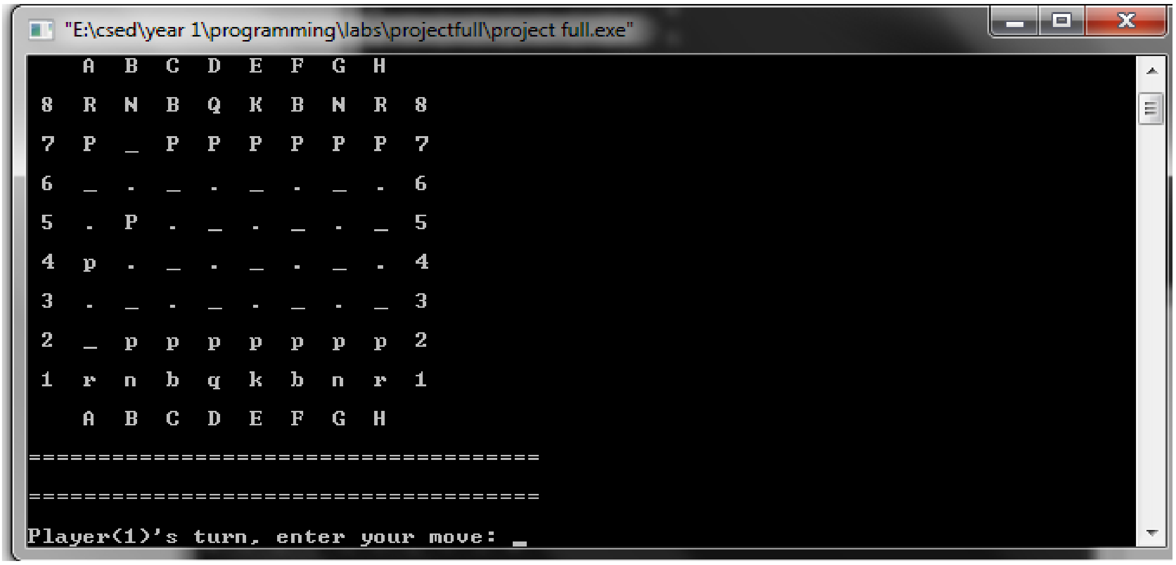
Second player:B7B5


"E:\csed\year 1\programming\labs\projectfull\project full.exe"

```
    A   B   C   D   E   F   G   H
8   R   N   B   Q   K   B   N   R   8
7   P   P   P   P   P   P   P   P   7
6   _   .   _   .   _   .   _   .   6
5   .   _   .   _   .   _   .   _   5
4   p   .   _   .   _   .   _   .   4
3   .   _   .   _   .   _   .   _   3
2   _   p   p   p   p   p   p   p   2
1   r   n   b   q   k   b   n   r   1
    A   B   C   D   E   F   G   H
=====================================
=====================================
Player(2)'s turn, enter your move: B7B5
```

How   do we enter a move?

m


"E:\csed\year 1\programming\labs\projectfull\project full.exe"

```
    A   B   C   D   E   F   G   H
8   R   N   B   Q   K   B   N   R   8
7   P   _   P   P   P   P   P   P   7
6   _   .   _   .   _   .   _   .   6
5   .   P   .   _   .   _   .   _   5
4   p   .   _   .   _   .   _   .   4
3   .   _   .   _   .   _   .   _   3
2   _   p   p   p   p   p   p   p   2
1   r   n   b   q   k   b   n   r   1
    A   B   C   D   E   F   G   H
=====================================
=====================================
Player(1)'s turn, enter your move: _
```

Pseudocode

Taking into consideration the the code is 3300 lines so that's the shortest pseudocode possible for each function

1) main function

```
 k, m, currentTurn, maxTurn,
  resume=1
  checkKing_1,checkKing_2,check_mate_King1,check_mate_King2,
  initialize_Board()
     copyBoard( Board, turns[currentTurn].board)
     copyPiece( piecesout1, turns[currentTurn].piecesout1)
     copyPiece( piecesout2, turns[currentTurn].piecesout2)
     turns[currentTurn].k = k
     turns[currentTurn].m = m
     currentTurn++
     maxTurn = currentTurn
  displayPlatform()
  while(resume)
               if(player==1)
        checkKing_1=CheckKing_1(King1_row(),King1_column())
        if(checkKing_1==1)
           check_mate_King1=Check_mate_king1()
           if(!check_mate_King1)
               display("Player(1)-Check !!")
            End if
           else
              display("Check_mate player(1)!\nplayer(2) wins")
              break
         end if
        else
           check_mate_King1=Check_mate_king1()
           if(check_mate_King1)
              display("This game is draw by Stale_mate !!")
              break
            end if
         end if
     end if
```

```
else
    checkKing_2=CheckKing_2(King2_row(),King2_column())
    if(checkKing_2==1)
        check_mate_King2=Check_mate_king2()
        if(!check_mate_King2)
            display("Player(2)-Check !!")
        else
            display("Check_mate player(2)!\nplayer(1) wins")
            break
    end if
    else
        check_mate_King2=Check_mate_king2()
        if(check_mate_King2)
            display("This game is draw by Stale_mate !!")
            break
      end if
   end if
end if
display("Player(%d)\'s turn, enter your move: ",player)
get input()
                                if(input=="save")
    saveGame()
else if(input=="load")
    loadGame()
else if(input=="undo")
    undo()
 end if
else if(input==redo)
    redo()
else
    moveValidation()
    displayPlatform()
                                end if
    //SWITCH PLAYER:
    if(player==1)
        player++
    else
        player=1
      end if
   end while
```

2)save function:

```
 saveGame()

   writeToFile(Board)
   writeToFile (piecesout1)
   writeToFile(piecesout2)
   writeToFile(k)
   writeToFile(m)
```

3) load function

```
 loadGame()
   writeToFile(Board)
   writeToFile (piecesout1)
   writeToFile(piecesout2)
   writeToFile(k)
   writeToFile(m)
```

4) undo function:

```
 undo()

   if( currentTurn > 0)

      currentTurn--
      copyBoard(turns[currentTurn].board, Board)
      copyPiece(turns[currentTurn].piecesout1,piecesout1)
      copyPiece(turns[currentTurn].piecesout2, piecesout2)
      k = turns[currentTurn].k
      m = turns[currentTurn].m
      if(player==1)
         player++
      else
         player=1
      end if
      displayPlatform()

   else
```

```
    displayPlatform();
    display("You can't undo any further")
  end if
```

5) redo function:

```
redo()

  if( currentTurn < maxTurn-1)

    currentTurn++
    copyBoard(turns[currentTurn].board, Board)
    copyPiece(turns[currentTurn].piecesout1,piecesout1)
    copyPiece(turns[currentTurn].piecesout2, piecesout2)
    k = turns[currentTurn].k
    m = turns[currentTurn].m
    if(player==1)
      player++
    else
      player=1
    end if
    displayPlatform()

  else
    displayPlatform();
    display("You can't redo any further.\n");
  end if
```

6) Move validation function:

```
 moveValidation()

label:
  ;
  int a,b,c,move,check
```

```
   col_1 = check_col_1(input[0])
   row_1 = check_row_1(input[1])
   col_2 = check_col_2(input[2])
   row_2 = check_row_2(input[3])

   a = (isEmpty(Board[row_1][col_1]==)|| (row_1==row_2 && col_1==col_2))

   b = (player==1 && !isEmpty(Board[row_1][col_1]==))
   c = (player==2 && !isEmpty(Board[row_1][col_1]==))

   if(player==1)
      move = Apply_move(row_1,col_1,row_2,col_2)
      check=CheckKing_1(King1_row(),King1_column())
   else
      move = Apply_move(row_1,col_1,row_2,col_2)
      check=CheckKing_2(King2_row(),King2_column())
 end if
   if(a==0 && b==0 && c==0 && check==0)
      move = Apply_move(row_1,col_1,row_2,col_2)
   else
      display("Your input is incorrect, please enter a valid move: ")
      get input()
      if(input=="save")
         saveGame()
      else if(input=="load")
         loadGame()
      else if(input=="undo")
         undo()
      else if(input=="redo")
         redo()
      end if
   end if
      displayPlatform()

   while(col_1==0 || row_1==0 || col_2==0 || row_2==0 || a || b || c ||
move==0)

      display("Your input is incorrect, please enter a valid move: ")
```

```
            get input()
        if(input==save)
            saveGame()
        else if(input==load)
            loadGame()
        else if(input==undo)
            undo()
        else if(input==redo)
            redo()
        else
            col_1 = check_col_1(input[0])
            row_1 = check_row_1(input[1])
            col_2 = check_col_2(input[2])
            row_2 = check_row_2(input[3])
            move = Apply_move(row_1,col_1,row_2,col_2)
        end if
end while
```

7) Check function:

this function is for player one and it is the same for player 2 with a little change in the indexes of the rows and columns

```
     CheckKing_1( x, y)
     Check,
    //ROOK OR QUEEN ON THE RIGHT
     check_1=0,    found=0;
    for  i=y+1 to 8   && !found && !check_1&& i++
        if(!isEmpty(Board[x][i]))
            if(Board[x][i]=='R' || Board[x][i]=='Q')
                check_1=1
            else
                found=1
                                                    end if
                                        end if
                                  end for
    //ROOK OR QUEEN ON THE LEFT
    check_2=0
    found=0
```

```
 for i=y-1 to 1 && !found && !check_2 && i--

    if(!isEmpty(Board[x][i]))

       if(Board[x][i]=='R' || Board[x][i]=='Q')
          check_2=1
       else
          found=1
                                     end if
                           end if
                      end for



 //ROOK OR QUEEN UP
 check_3=0
 found=0
 for i=x-1 to 1 && !found && !check_3 && i--
    if(!isEmpty(Board[i][y]))
       if(Board[i][y]=='R' || Board[i][y]=='Q')
          check_3=1
       else
          found=1
                                     end if
                           end if
                      end for
 //ROOK OR QUEEN DOWN
check_4=0
 found=0
 for i=x+1 to 8 && !found && !check_4 &&i++
    if(!isEmpty(Board[i][y]))
       if(Board[i][y]=='R' || Board[i][y]=='Q')
          check_4=1
       else
          found=1
                                     end if
                           end if
                      end for
 //BISHOP OR QUEEN LOWER-RIGHT CORNER
```

```
found=0
check_5=0
j=y+1
for(i=x+1 to 8 && !found && !check_5 && i++
    if(!isEmpty(Board[i][j]))
      if(Board[i][j]=='B' || Board[i][j]=='Q')
                                      check_5=1
                               else
                                      found=1
                               end if
                        end if
  j++
               end for
  //BISHOP OR QUEEN UPPER-LEFT CORNER
found=0
 check_6=0
j=y-1
for(i=x-1 to 1 && !found && !check_6 &&i--
  if(!isEmpty(Board[i][j]))
    if(Board[i][j]=='B' || Board[i][j]=='Q')
      check_6=1
    else
      found=1
                        end if
                  end if
  j--
end for

//BISHOP OR QUEEN LOWER-LEFT CORNER
found=0
check_7=0
j=y-1
for i=x+1 to 8  && !found && !check_7 && i++
  if(!isEmpty(Board[i][j]))
    if(Board[i][j]=='B' || Board[i][j]=='Q')
      check_7=1
    else
      found=1
```

```
                          end if
                    end if
    j--
  end for
  //BISHOP OR QUEEN UPPER-RIGHT CORNER
  found=0
  check_8=0
  j=y+1
  for i=x-1 to 1  && !found && !check_8 && i--)
    if(!isEmpty(Board[i][j]))
      if(Board[i][j]=='B' || Board[i][j]=='Q')
        check_8=1
      else
        found=1
                          end if
                    end if
                    j++
                 end for
  //PAWN THREATENING PLAYER(1)'S KING
   check_9=0
  if(Board[x-1][y-1]=='P' || Board[x-1][y+1]=='P')
    check_9=1
  //KNIGHTS
   check_10=0
  if(knightThreat(x,y))
    check_10=1
  //CHECK_TEST
  check =
check_1||check_2||check_3||check_4||check_5||check_6||check_7||check_
8||check_9||check_10
  return check
```

8) Checkmate function:

this function is for player one and it is the same for player 2 with a little change in the indexes of the rows and columns and using isLower instead of isUpper

Check_mate_king1()

```
 x,y,i1,j1,i2,j2,
x=King1_row()
y=King1_column()
check_1,out_1
if((x-1)>0 && (x-1)<8 && isUpper(Board[x-1][y]))
   check_1=CheckKing_1(x-1,y)
else
   out_1=1
                    end if

check_2,out_2
if((x+1)>1 && (x+1)<9 && isUpper(Board[x+1][y]))
   check_2=CheckKing_1(x+1,y)
else
   out_2=1
end if
check_3,out_3
if((y-1)>0 && (y-1)<8 && isUpper(Board[x][y-1]))
   check_3=CheckKing_1(x,y-1)
else
   out_3=1
end if
int check_4,out_4
if((y+1)>1 && (y+1)<9 && isUpper(Board[x][y+1]))
   check_4=CheckKing_1(x,y+1)
else
   out_4=1
end if
 check_5,out_5
if((x+1)>1 && (x+1)<9 && (y+1)>1 && (y+1)<9 && isUpper(Board[x+1][y+1]))
   check_5=CheckKing_1(x+1,y+1)
else
   out_5=1
end if
check_6,out_6
if((x-1)>0 && (x-1)<8 && (y-1)>0 && (y-1)<8 && isUpper(Board[x-1][y-1]))
   check_6=CheckKing_1(x-1,y-1)
```

```
      else
         out_6=1
      end if
      check_7,out_7
      if((x+1)>1 && (x+1)<9 && (y-1)>0 && (y-1)<8 && isUpper(Board[x+1][y-1]))
         check_7=CheckKing_1(x+1,y-1)
      else
         out_7=1
         end if
       check_8,out_8
      if((x-1)>0 && (x-1)<8 && (y+1)>1 && (y+1)<9 && isUpper(Board[x-1][y+1]))
         check_8=CheckKing_1(x-1,y+1)
      else
         out_8=1
      end if
      check_around =
   (check_1||out_1)&&(check_2||out_2)&&(check_3||out_3)&&(check_4||out_4
   )&&(check_5||out_5)&&(check_6||out_6)&&(check_7||out_7)&&
   (check_8||out_8)
 if(1)
       for i=0 to 9 &&i++
         for j=0 to 9 &&j++
         Alternative[i][j]=Board[i][j]
         End for
        End for
         simulation=1
          temp
          move, check_in_simultion,
          safe=0
              for i1=1 to 8 &&!safe && i1++
                  for  j1=1 to 8 &&!safe &&j1++

                     if(Board[i1][j1]=='p' || Board[i1][j1]=='n' || Board[i1][j1]=='r' ||
                        Board[i1][j1]=='b' || Board[i1][j1]=='q')

                           for i2=1 to 8 &&!safe && i2++
                                for  j2=1 to 8 &&!safe &&j2++
                                  temp=Board[i2][j2]
```

```
                          move=Apply_move(i1,j1,i2,j2)
                              if(move)
                              check_in_simultion=CheckKing_1(x,y)
                                  if(!check_in_simultion)
                                     safe=1
                                     for i=0 to 9 &&i++
                                   for j=0 to 9&& j++
                                   Board[i][j]=Alternative[i][j]
                                     End for
                                     End for
                                   return 0
                                else

                               Board[i1][j1]=Board[i2][j2]
                               Board[i2][j2]=temp
                                  For i=0 to 9&&i++
                                   for j=0 to 9 && j++
                                     Board[i][j]=Alternative[i][j]
                                   End for
                                  End for

                                End if
                             End if
                           End for
                         End for
                       End if
                    End for
                   End for
         if(safe==0)
          return 1
         end if
  else
return 0
 end if
 end
```

9) **Locating the king**:

the next 2 functions are for player1 but their equivalent for player 2 are the same with a little change in the indexing of the rows and columns

- King1_row()

```
    //LOCATING THE ROW OF PLAYER(1)'S KING
    king=0;
//x_y DETERMINE THE EXACT LOCATION OF KING ON THE BOARD
    For i=1 to 8 && !king&& i++
        For j=1 to 8 && !king&& j++

            if(Board[i][j]=='k')
                king=1
        end if
     end for
  end for
    return(i-1)
```

- King1_column()

```
    //LOCATING THE COLUMN OF PLAYER(1)'S KING
    int king=0;
 //x_y DETERMINE THE EXACT LOCATION OF KING ON THE BOARD
    For i=1 to 8 && !king &&i++
        For j=1 to 8 && !king &&j++

            if(Board[i][j]=='k')
                king=1
        end if
       end for
     end for
    return(j-1)
```

10) APPLYING THE CORRECT MOVE:

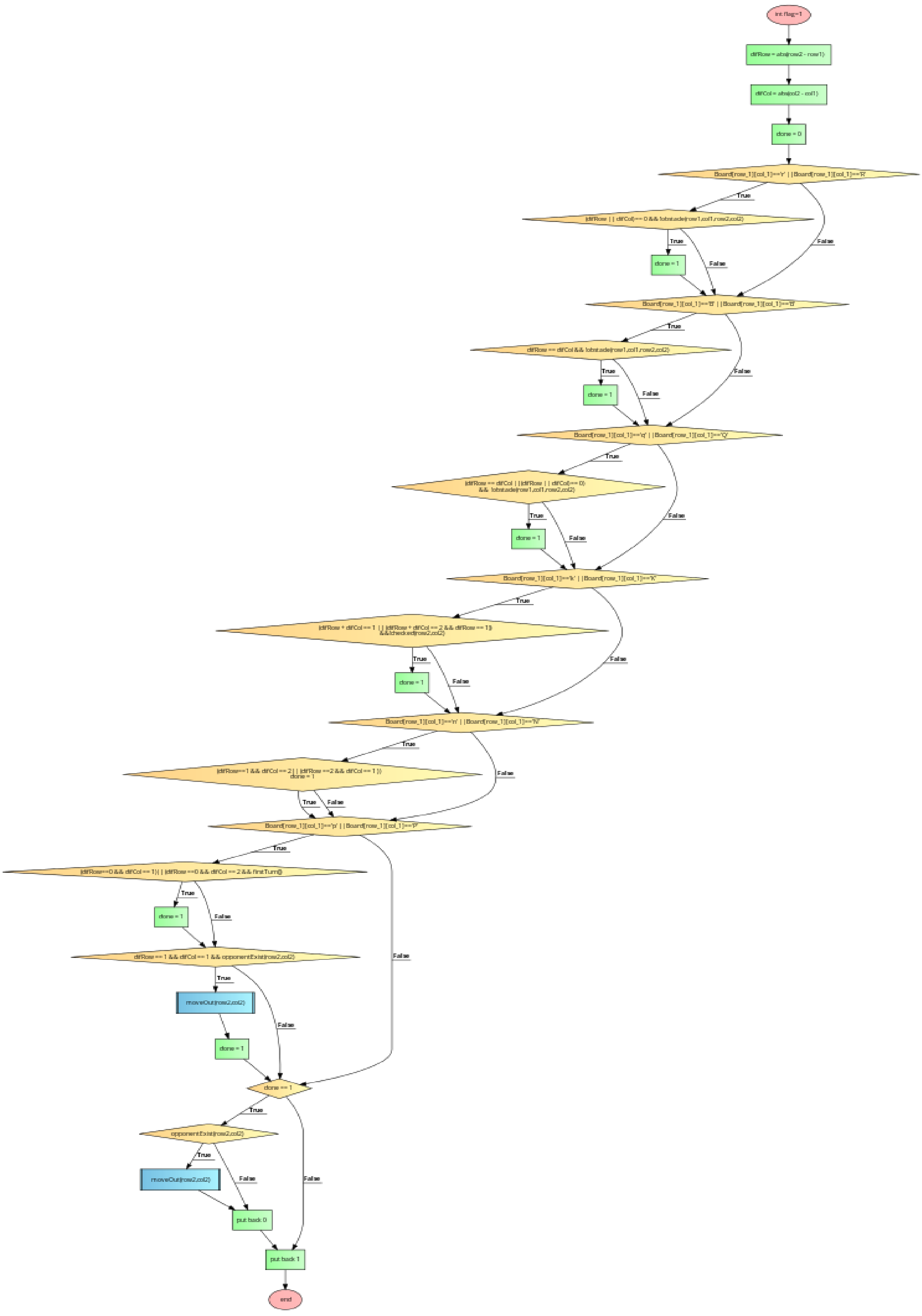
Apply_move(row_1,col_1,row_2,col_2)

```
   int flag=1
difRow = abs(row2 - row1)
difCol = abs(col2 - col1)
done = 0
if( Board[row_1][col_1]=='r' ||Board[row_1][col_1]=='R')
        if( (difRow || difCol) == 0 && !obstacle(row1,col1,row2,col2))
                                        done = 1
         end if
end if
if( Board[row_1][col_1]=='B' ||Board[row_1][col_1]=='B')
       if( difRow == difCol && !obstacle(row1,col1,row2,col2))
                                        done = 1
        end if
end if
if( Board[row_1][col_1]=='q' ||Board[row_1][col_1]=='Q')
     if( (difRow == difCol ||(difRow || difCol) == 0)
                && !obstacle(row1,col1,row2,col2))
                                        done = 1
       end if
 end if



if( Board[row_1][col_1]=='k' ||Board[row_1][col_1]=='K')
          if( (difRow + difCol == 1 || (difRow + difCol == 2 && difRow == 1))
             &&!checked(row2,col2))
                                        done = 1
          end if
end if
if( Board[row_1][col_1]=='n' ||Board[row_1][col_1]=='N')
         if( (difRow==1 && difCol == 2|| (difRow ==2 && difCol == 1 ) )
                                        done = 1
        end if
end if
if( Board[row_1][col_1]=='p' ||Board[row_1][col_1]=='P')
      if( (difRow==0 && difCol == 1)|| (difRow ==0 && difCol == 2 && firstTurn()) )
                                        done = 1
      end if
```

```
        if(difRow == 1 && difCol == 1 && opponentExist(row2,col2))
                        moveOut(row2,col2)
                        done = 1
         end if
end if
if(done == 1)
    if(opponentExist(row2,col2))
        moveOut(row2,col2)
     end if
    return 0
end if
return 1
end
```

Flowcharts

1)Apply move function:

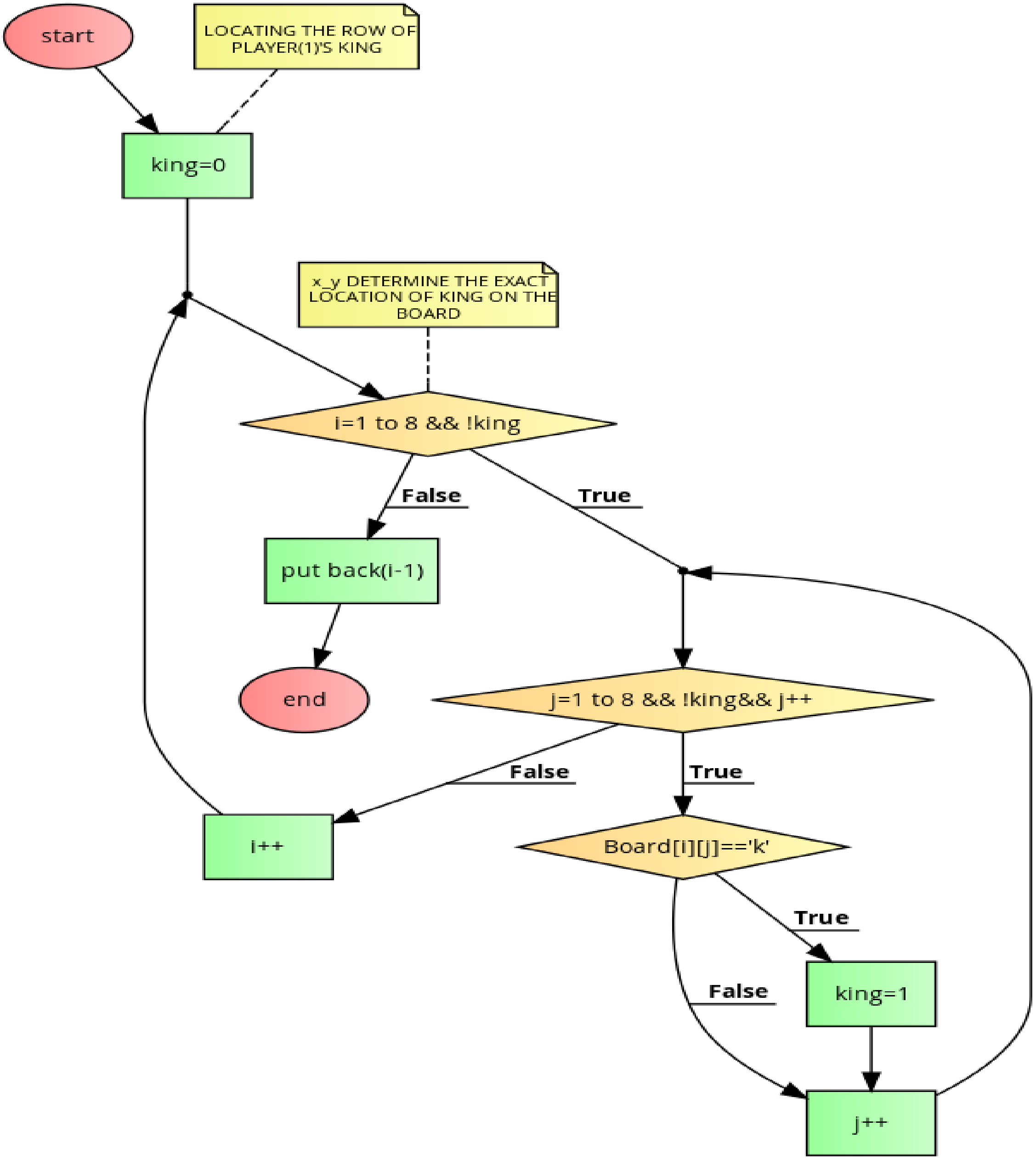

int flag=1
difRow = abs(row2 - row1)
difCol = abs(col2 - col1)
done = 0
Board[row_1][col_1]=='r' | | Board[row_1][col_1]=='R'
True
False
(difRow | | difCol)== 0 && !obstacle(row1,col1,row2,col2)
True
False
done = 1
Board[row_1][col_1]=='b' | | Board[row_1][col_1]=='B'
True
False
difRow == difCol && !obstacle(row1,col1,row2,col2)
True
False
done = 1
Board[row_1][col_1]=='q' | | Board[row_1][col_1]=='Q'
True
False
(difRow == difCol | | (difRow | | difCol)== 0)
&& !obstacle(row1,col1,row2,col2)
True
False
done = 1
Board[row_1][col_1]=='k' | | Board[row_1][col_1]=='K'
True
False
(difRow + difCol == 1 | | (difRow + difCol == 2 && difRow == 1))
&&!checked(row2,col2)
True
False
done = 1
Board[row_1][col_1]=='n' | | Board[row_1][col_1]=='N'
True
False
(difRow==1 && difCol == 2 | | (difRow ==2 && difCol == 1))
done = 1
True
False
Board[row_1][col_1]=='p' | | Board[row_1][col_1]=='P'
True
False
(difRow==0 && difCol == 1)| | (difRow ==0 && difCol == 2 && firstTurn())
True
False
done = 1
difRow == 1 && difCol == 1 && opponentExist(row2,col2)
True
False
moveOut(row2,col2)
done = 1
done == 1
True
False
opponentExist(row2,col2)
True
False
moveOut(row2,col2)
put back 0
put back 1
end

2)Locating the king:

the next 2 functions are for player1 but their equivalent for player 2 are the same with a little change in the indexing of the rows and columns

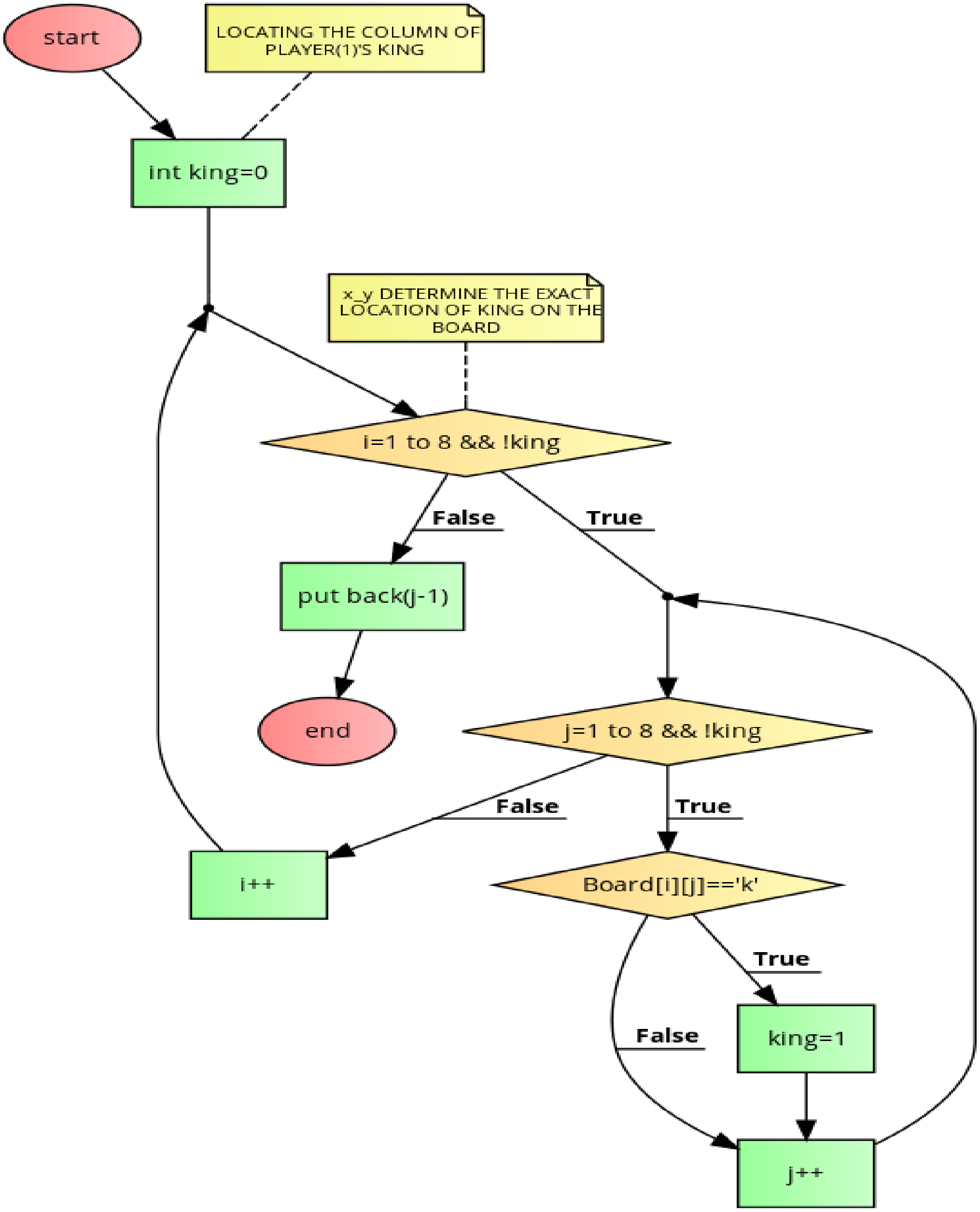
- King1_row()

  //LOCATING THE ROW OF PLAYER(1)'S KING



- King1_column()

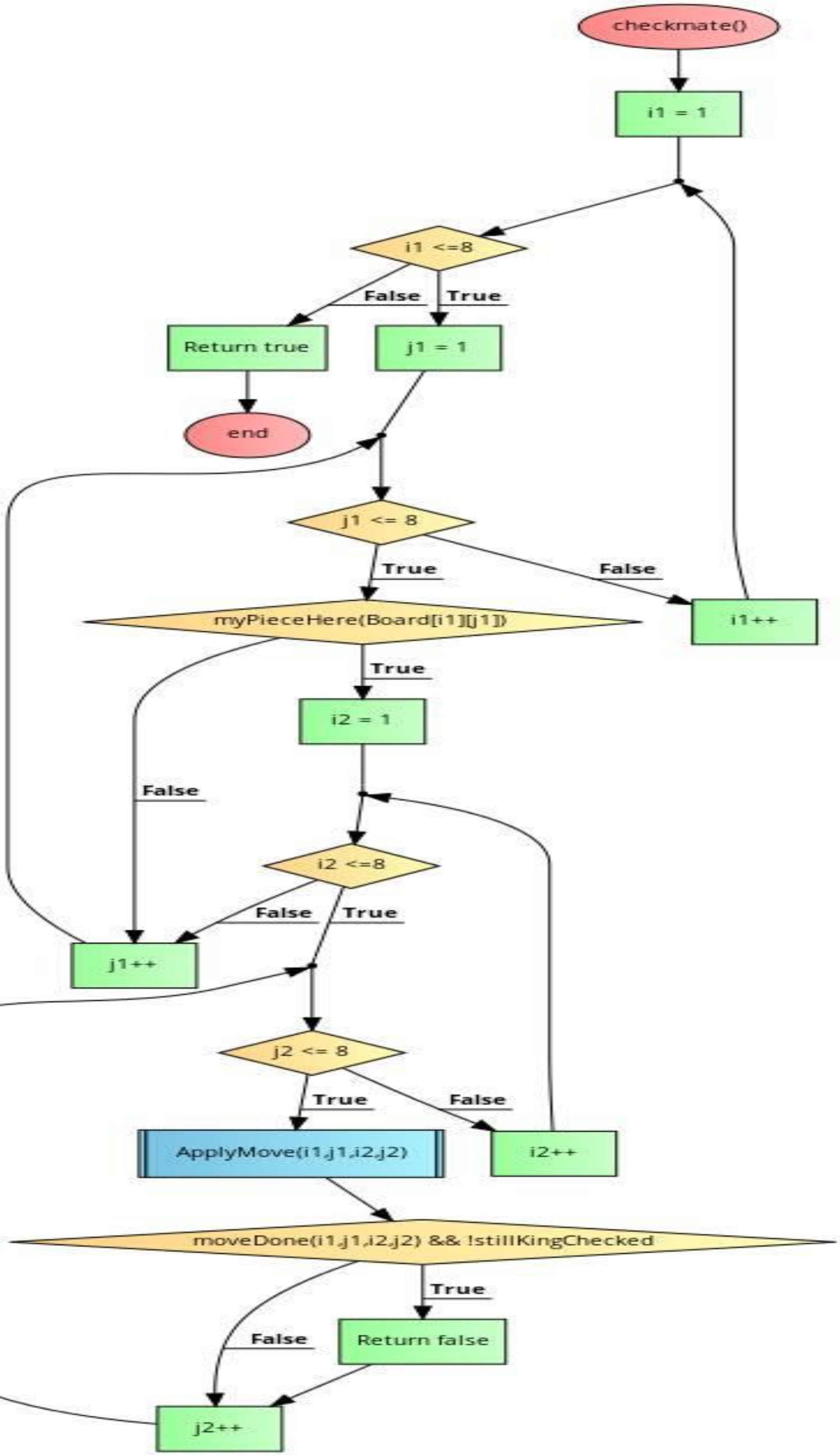
//LOCATING THE COLUMN OF PLAYER(1)'S KING



3) Checkmate function:

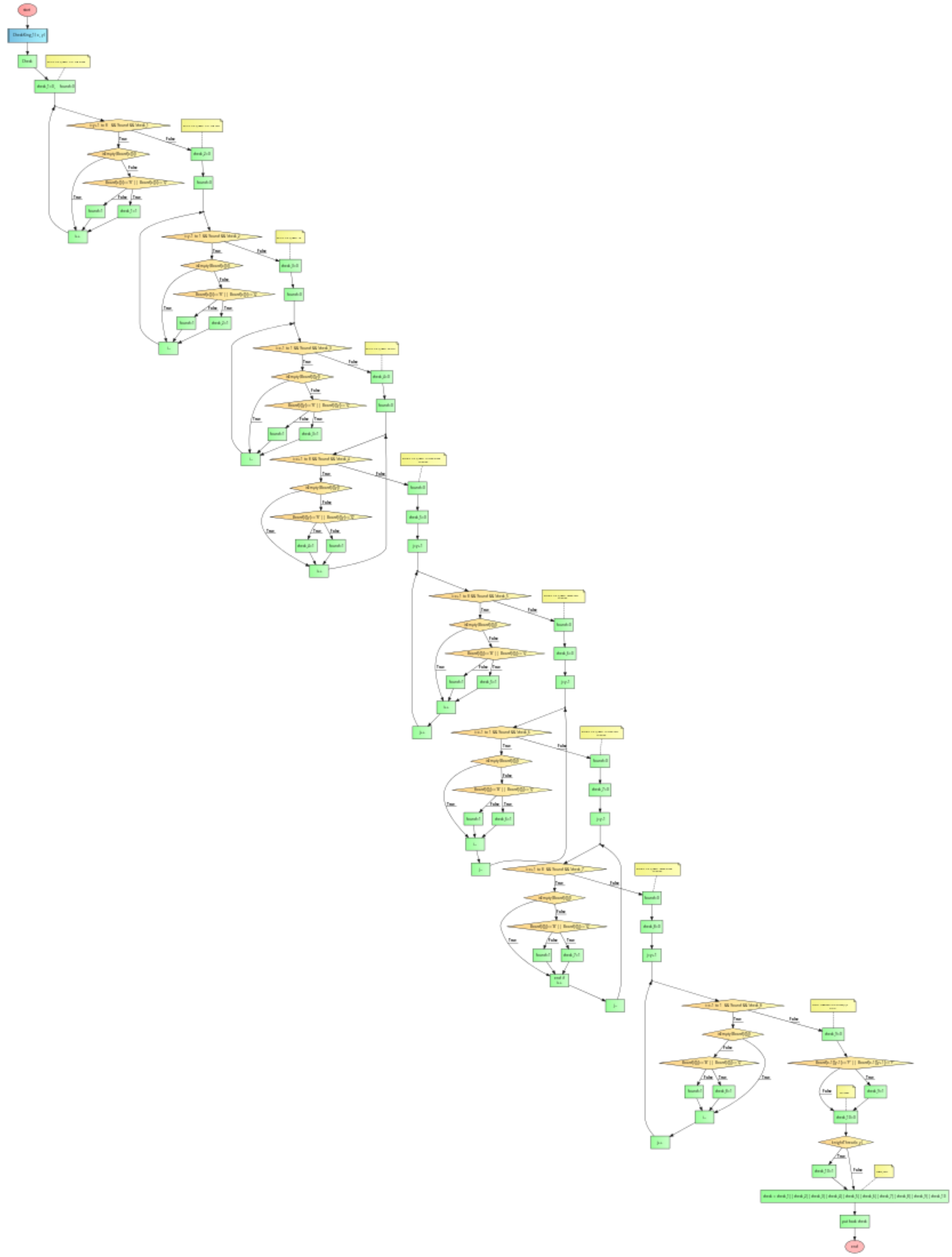
this function is for player one and it is the same for player 2 with a little change in the indexes of the rows and columns and using isLower instead of isUpper

Check_mate_king1()

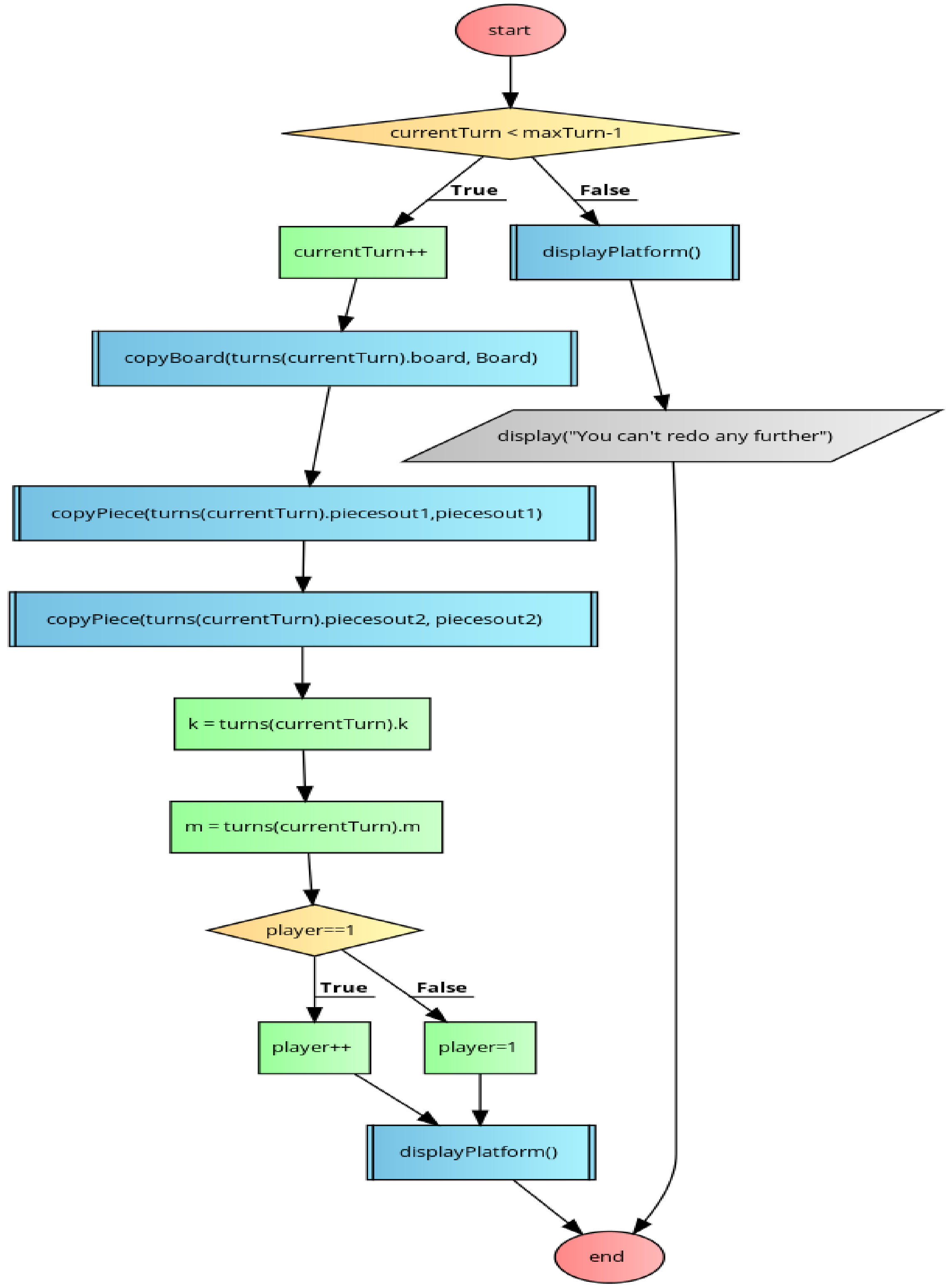

checkmate()
i1 = 1
i1 <=8
False
True
Return true
j1 = 1
end
j1 <= 8
True
False
myPieceHere(Board[i1][j1])
i1++
True
i2 = 1
False
i2 <=8
False
True
j1++
j2 <= 8
True
False
ApplyMove(i1,j1,i2,j2)
i2++
moveDone(i1,j1,i2,j2) && !stillKingChecked
True
False
Return false
j2++

4)Check function:

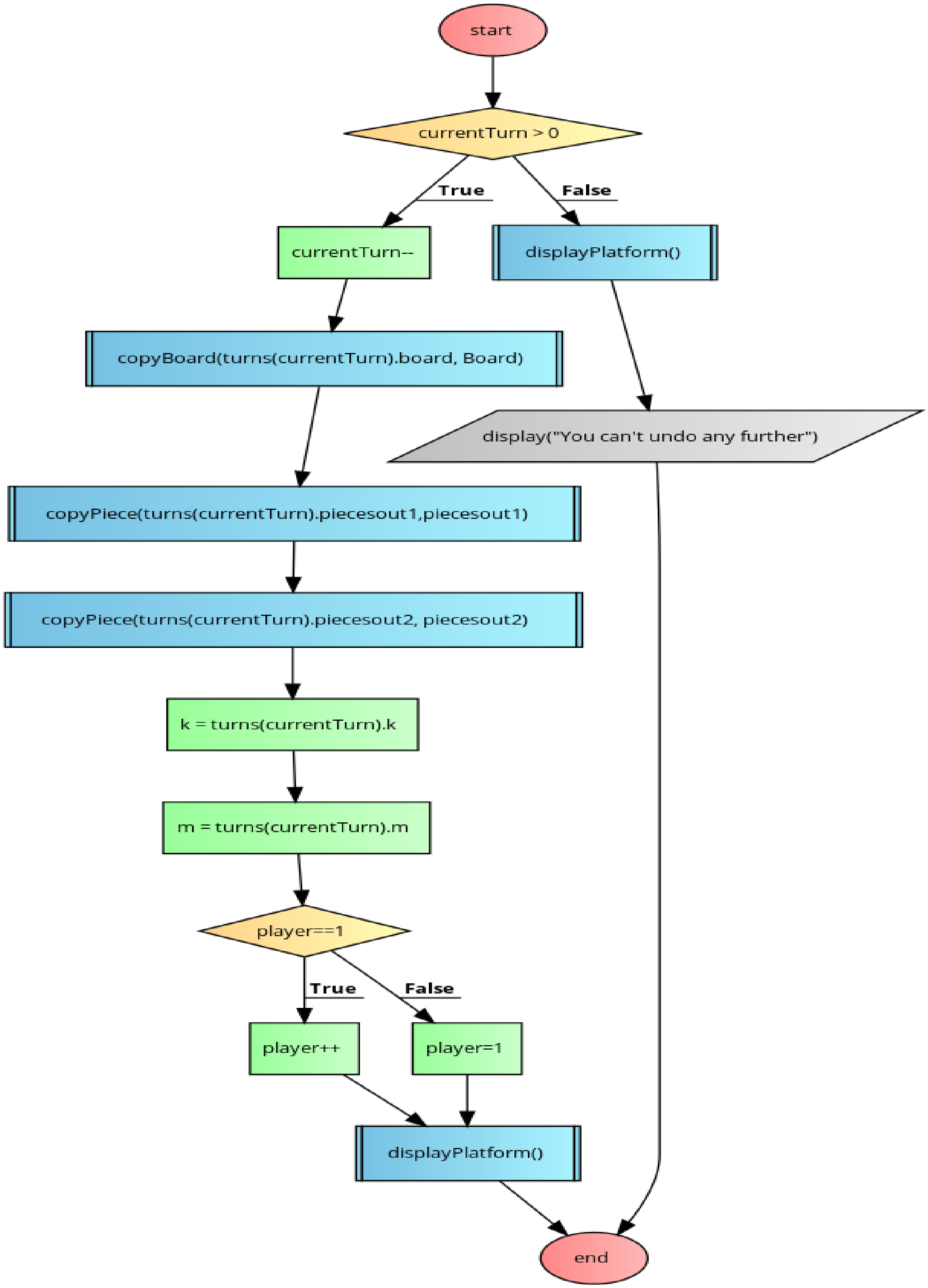
this function is for player one and it is the same for player 2 with a little change in the indexes of the rows and columns



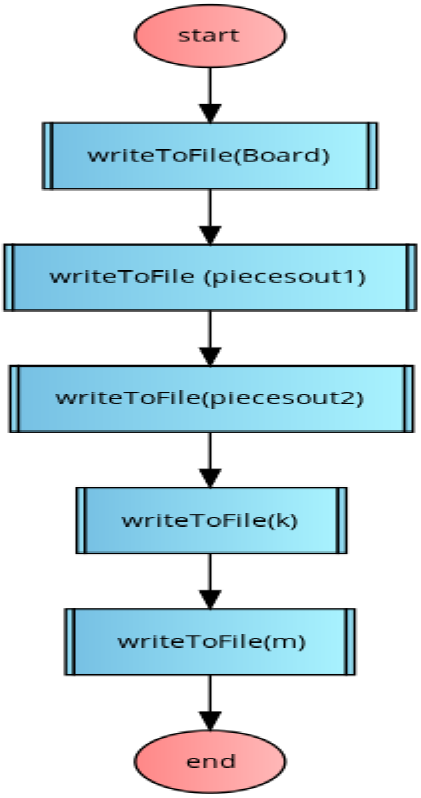
5)Redo function:


start
currentTurn < maxTurn-1
True
False
currentTurn++
displayPlatform()
copyBoard(turns(currentTurn).board, Board)
display("You can't redo any further")
copyPiece(turns(currentTurn).piecesout1,piecesout1)
copyPiece(turns(currentTurn).piecesout2, piecesout2)
k = turns(currentTurn).k
m = turns(currentTurn).m
player==1
True
False
player++
player=1
displayPlatform()
end

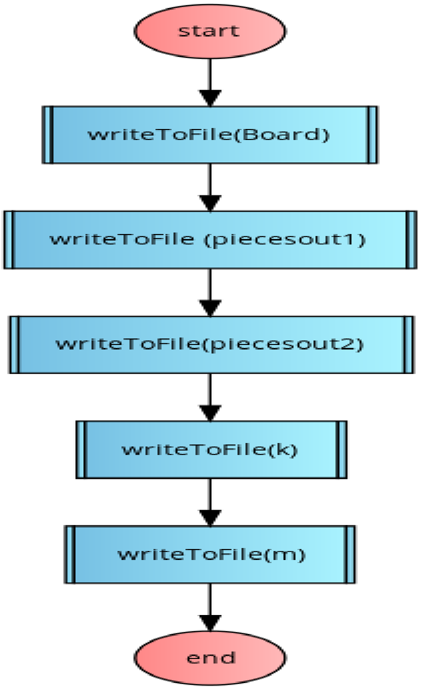
6)Undo function:


start
currentTurn > 0
True
False
currentTurn--
displayPlatform()
copyBoard(turns(currentTurn).board, Board)
display("You can't undo any further")
copyPiece(turns(currentTurn).piecesout1,piecesout1)
copyPiece(turns(currentTurn).piecesout2, piecesout2)
k = turns(currentTurn).k
m = turns(currentTurn).m
player==1
True
False
player++
player=1
displayPlatform()
end

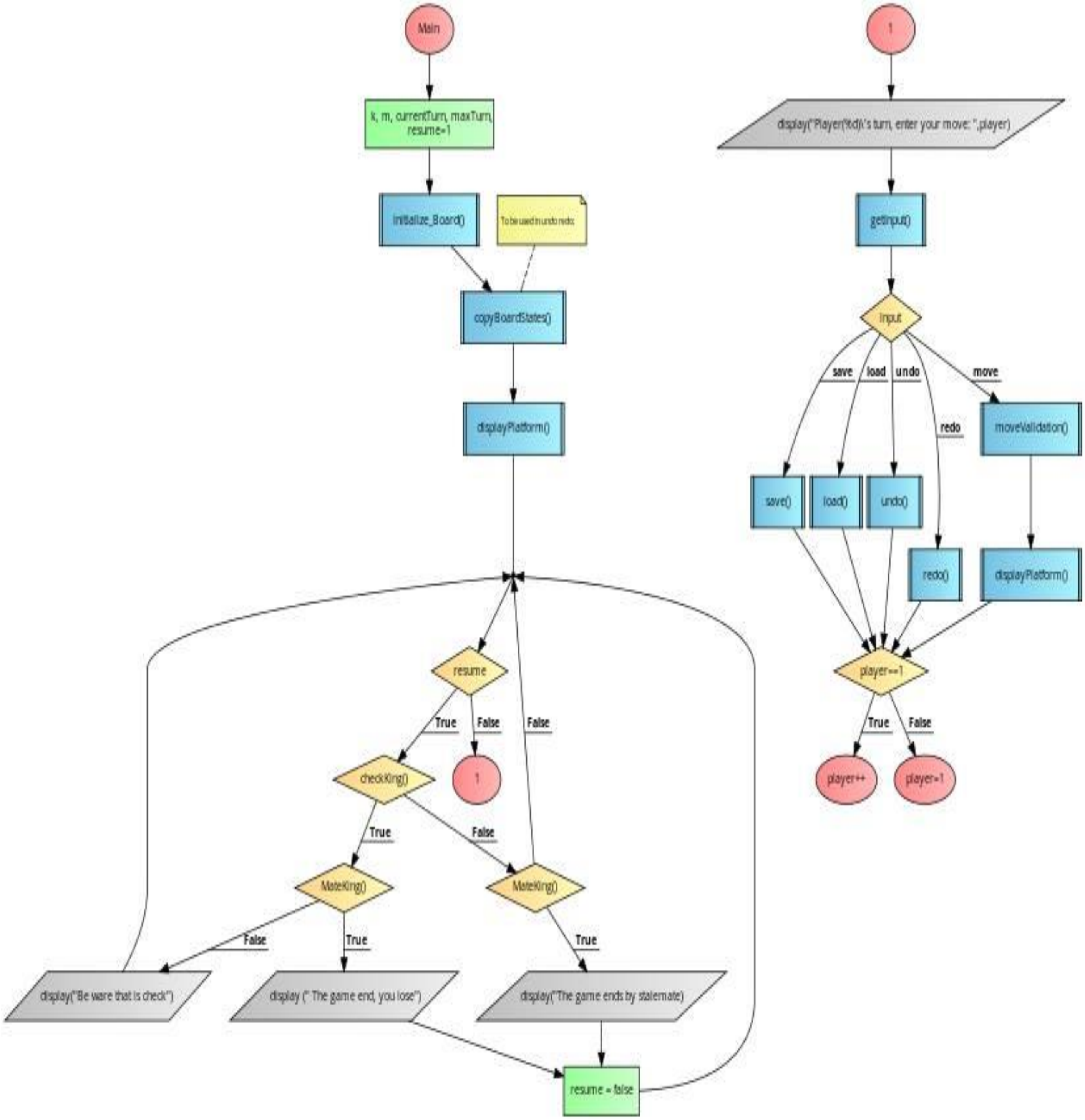
7)Save function:



8)Load function:



9)Main function:


Main
k, m, currentTurn, maxTurn, resume=1
initialize_Board()
To be used in undo redo;
copyBoardStates()
displayPlatform()
resume
True
False
False
checkKing()
1
True
False
MateKing()
MateKing()
False
True
True
display("Be ware that is check")
display (" The game end, you lose")
display("The game ends by stalemate)
resume = false
1
display("Player(%d)\'s turn, enter your move: ",player)
getInput()
Input
save
load
undo
redo
move
moveValidation()
save()
load()
undo()
redo()
displayPlatform()
player==1
True
False
player++
player=1