Binus International

Data Structures | Object Oriented Programming

Final Project Report

Chess Engine with FEN

Submitted by

Rafie Mustika Ramasna : 2802522815

Jovanney Rafael Husni : 2802523105

Thristan Widjaja : 2802503690

Submitted To : FERI SETIAWAN, S.Kom, M.Eng., Ph

TABLE OF CONTENTS

1. Background

In modern software development, object-oriented programming (OOP) and data structures play a crucial role in building efficient and scalable systems. One of the most challenging applications of these principles is chess engine development, which requires advanced computational thinking, algorithm design, and optimization techniques. Chess engines evaluate millions of positions per second, making them an excellent testbed for complex algorithms and data structures [1].

Chess has long been recognized as a powerful tool for improving cognitive skills. Research shows that playing chess enhances critical thinking, problem-solving, and memory [2]. A study conducted in Zaire found that students who played chess demonstrated significant improvements in spatial, numerical, and verbal abilities compared to a control group [3]. Similarly, a study in Belgium revealed that chess-playing students exhibited greater cognitive development, as measured by Piaget's tests, and outperformed their peers in standardized testing [4]. Further research in Hong Kong found that students who played chess improved their math and science test scores by 15% [5]. Another four-year study in Pennsylvania showed that students who played chess consistently outperformed those in other thinking development programs, as measured by critical thinking and creativity tests [6].

Beyond its cognitive benefits, chess is particularly effective in improving attention span. A study on chess and ADHD found that playing chess helps individuals maintain focus for extended periods, improving sustained attention and reducing impulsivity [7]. Another study on computational thinking and chess revealed that structured problem-solving in chess aligns closely with algorithmic thinking, reinforcing logical reasoning and decision-making skills [8]. Research on AI-assisted chess training further supports the idea that chess engines enhance human decision-making and pattern recognition, making them valuable learning tools [9].

From a programming perspective, chess engine development involves key OOP principles such as encapsulation, inheritance, and polymorphism. These principles allow efficient modeling of chess pieces, the board, and game rules. Additionally, data structures play a central role in chess engines. Trees are used for move generation and game state evaluation, with minimax trees and game trees forming the foundation of decision-making processes. Hash tables, particularly transposition tables, store previously evaluated positions to reduce redundant calculations. Priority queues help optimize move ordering, ensuring that the most promising moves are evaluated first, improving efficiency. Search algorithms such as minimax with alpha-beta pruning reduce the number of positions the engine needs to analyze, significantly increasing performance.

A crucial aspect of any robust chess engine is the effective representation and manipulation of game states. Each unique arrangement of pieces on the chessboard, along with relevant game metadata (like whose turn it is, castling rights, and en passant target squares), constitutes a distinct game state. The Forsyth-Edwards Notation (FEN) has emerged as the standard text-based format for describing a chess board position, providing a compact and portable way to capture a game's precise status [10]. However, as chess engines delve deeper into game trees, potentially exploring millions or billions of states, the efficiency with which these FEN strings and their corresponding board representations are stored, accessed, and modified becomes paramount. Inefficient data structure choices can lead to significant performance bottlenecks, resulting in slower runtime speeds and limiting the engine's ability to conduct thorough searches [11].

This issue of state management is not unique to chess. In various computational systems, the selection of an appropriate data structure is fundamental to achieving optimal performance. An effective data structure can make an application highly responsive to specific inputs and adept at handling large volumes of data, forming the backbone of efficient computer programs [12]. For chess engines, where milliseconds can dictate the depth of search and the quality of a move, optimizing the underlying data structures for board representation and lookup is critical. Previous research highlights how poor data structure choices can lead to substantial performance degradation, particularly in scenarios involving concurrent access or vast data sets, as observed in systems like Moodle's online quiz feature under high user load [13].

Given the complexity of chess and its deep connection to computational problem-solving, this project aims to address the challenge of efficient game state management within a chess engine context. By developing a chess engine in Java that utilizes FEN strings for board representation, a primary objective is to investigate and compare the performance implications of different data structures for storing and retrieving these game states, specifically focusing on **ArrayList, HashMap, and LinkedList**. This comparative analysis will not only enhance the understanding of how various data structures impact the runtime efficiency and memory footprint of a complex AI application but also contribute to the ongoing refinement of chess engine design and optimization [14]. The insights gained will highlight best practices for managing dynamic, high-volume data in algorithmic systems, offering valuable lessons applicable beyond the realm of game AI [15].

2. Problem Description

Chess is an ever evolving game as stated in the background and with that it's all the more important to facilitate the game in our application. Starting with the aspects which do not particularly relate with data structures, we are planning to create an application which allows two people to move their specific chess pieces as they are described in the game during their turn. This includes rules like castling, en passant, and regular piece taking.

With regards to data structures, the plan we have decided on is to implement a system where we can export and import a game up until a certain point from the very beginning and test the efficacy of an ArrayList, LinkedList, HashMaps in time and memory to find which data structure is the best.

To do this, we are going to implement something called FEN. FEN or Forsyth-Edwards Notation is a standard notation system created by journalist David Forsyth and expanded by Steven J. Edwards. This notation is a single ASCII string and contains all the data about a chess position at one exact moment.

After every move we will be adding a new FEN string and chaining them into the data structures stated above and we will be finding the space and time used when doing the following tasks in the engine:

1. Saving a new FEN string to the data structure after a move.
2. Exporting match until current point to save file.
3. Importing matches from a save file and loading onto the engine.
4. Searching for a particular moment in the game.

The space used can be seen from the memory usage of the function in the data structure and the time used can be seen from the average runtime speed. From this we are able to find the most suited data structure in this task to create our chess engine with FEN.

With regards to our Object Oriented Programming, we needed to find a way to convert Chess Pieces to regular strings and vice versa, the engine should also do the following:

1. Correctly move in accordance to the clicked Chess Piece
2. Have mouse functionality to click squares
3. Have appropriate buttons to navigate the functions.
4. Obviously, change FEN to pieces and vice versa

3. Solution Design

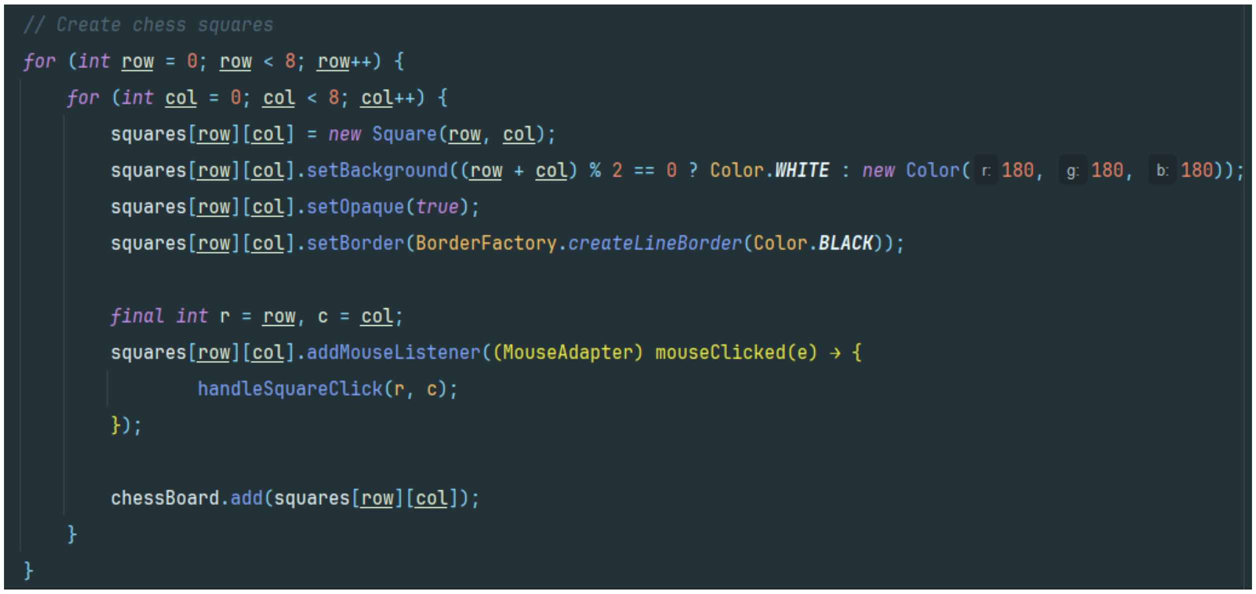
3.1 Function Descriptions

➔ **Initialization**

With regards to the functionality of the program, the first thing that we needed to do is actually place everything within a GUI environment. For this, we are using **Java Swing** as shown by our main class JavaApplication being a subclass of **JFrame**, allowing us to build our UI using panels.

When it comes to integrating FEN as well, we decided to make it so that the squares of the chessboard play a pivotal role when doing the logic. To do this, we first created a 2D array using another one of our classes *Squares*.

Within the squares are a variable which will describe their piece as well as 2 variables which would dictate their particular row and column within the 2D array. It is initialized through a nested for loop at the start of the application which will give it its particular row and column as shown below. It is then added to a GridLayout JPanel called ChessBoard where it will be displayed in the Application.



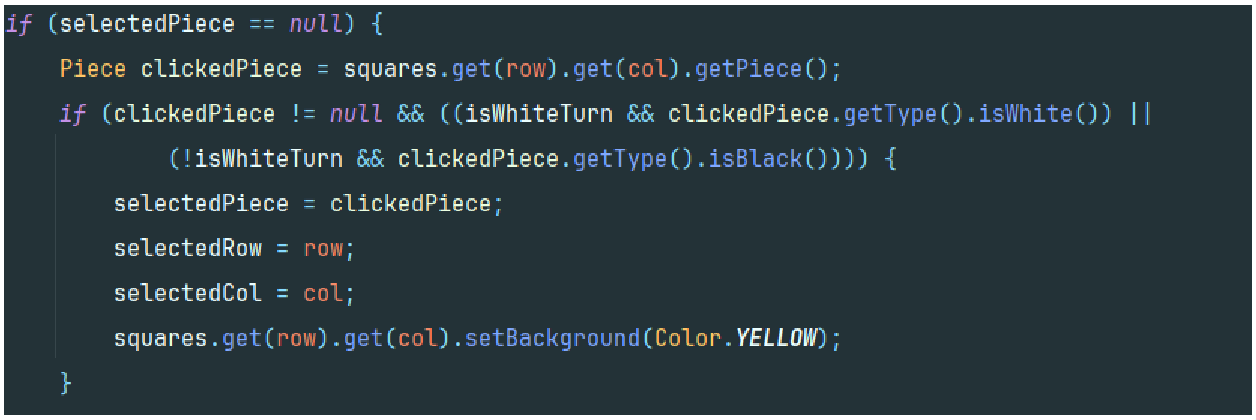
```
// Create chess squares
for (int row = 0; row < 8; row++) {
    for (int col = 0; col < 8; col++) {
        squares[row][col] = new Square(row, col);
        squares[row][col].setBackground((row + col) % 2 == 0 ? Color.WHITE : new Color( r: 180,  g: 180,  b: 180));
        squares[row][col].setOpaque(true);
        squares[row][col].setBorder(BorderFactory.createLineBorder(Color.BLACK));

        final int r = row, c = col;
        squares[row][col].addMouseListener((MouseAdapter) mouseClicked(e) → {
                handleSquareClick(r, c);
        });

        chessBoard.add(squares[row][col]);
    }
}
```

Figure 3.1.1 *Initializing the Squares and the UI*

➔ **Mouse Clicks**

As shown above here we added a MouseListener on the squares that runs the function HandleSquareClick(r, c), with r being row and c being column, whenever the mouse button is clicked on any given square. This is the primary way at which the player is able to interact with the board.

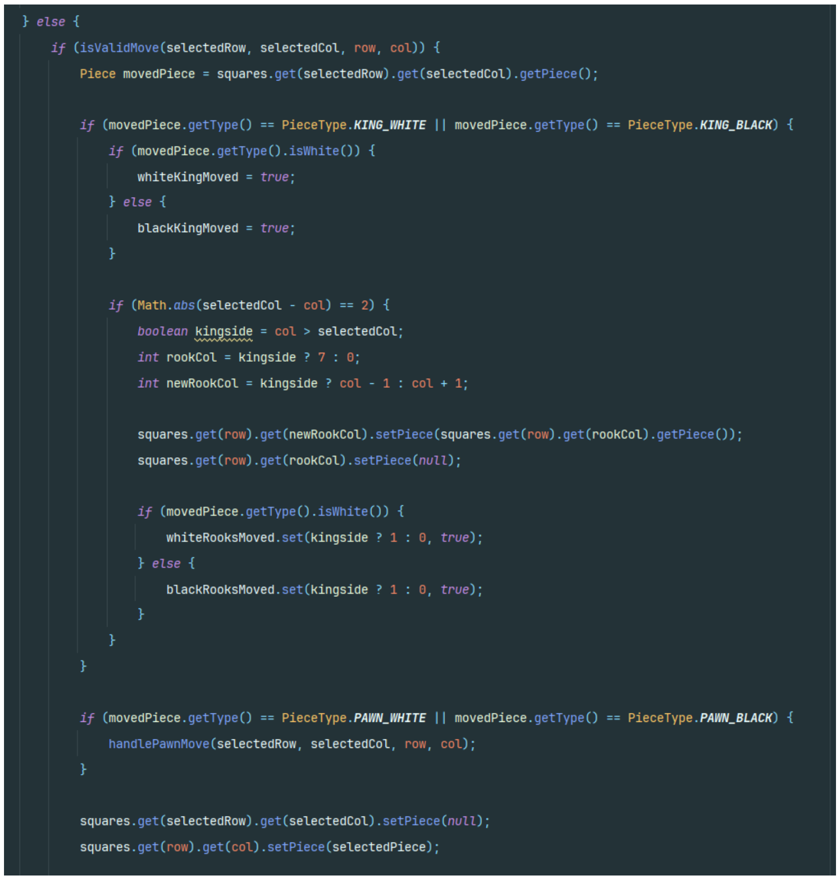
Looking into the function we can see that if there is no piece selected then it will attempt to retrieve the piece clicked by using getPiece at the square of that clicked row and column and then it will check again to see if that clicked piece is indeed not null and if the the isWhite boolean of the piece coincides with the isWhiteTurn boolean of the current moment of the game, then it will update the selectedPiece variable to the clickedPiece.



```java
if (selectedPiece == null) {
    Piece clickedPiece = squares.get(row).get(col).getPiece();
    if (clickedPiece != null && ((isWhiteTurn && clickedPiece.getType().isWhite()) ||
            (!isWhiteTurn && clickedPiece.getType().isBlack()))) {
        selectedPiece = clickedPiece;
        selectedRow = row;
        selectedCol = col;
        squares.get(row).get(col).setBackground(Color.YELLOW);
    }
```

Figure 3.1.2 *No selected piece / selecting for the first time*

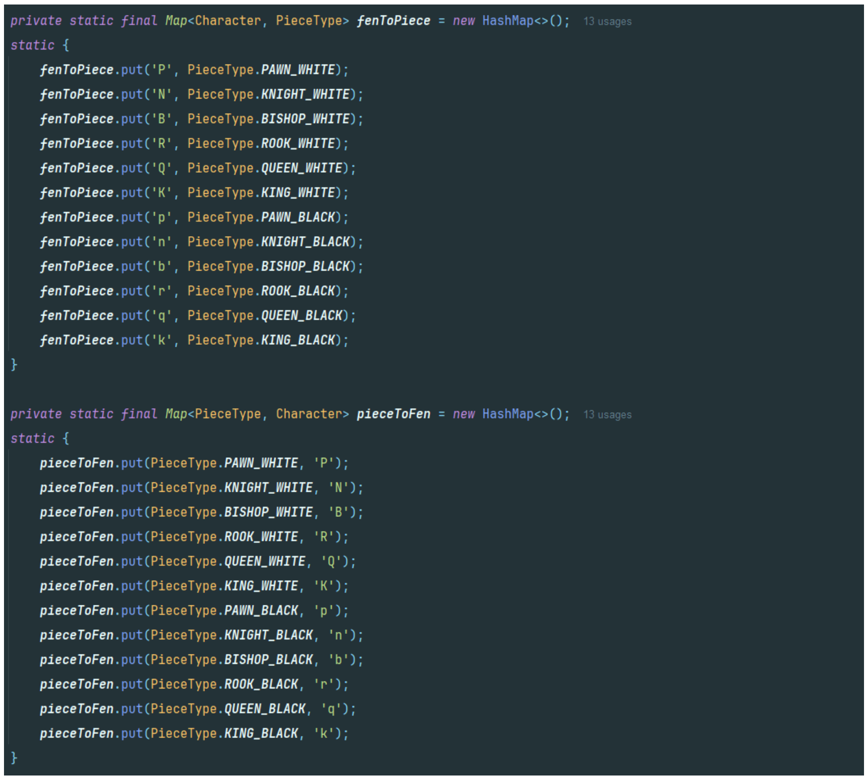
Now that it knows that there is a selected piece, it will check if the next click that you make will be valid, using IsValidMove, which we will cover more in-depth later. Then it will check for some extra things like if you're doing a double pawn move at the start, castling, or promoting and if those are false then it will get the row of the original piece, turn it to null and set the piece of the clicked square to the piece that was already selected. At the end it will add the move to the moveHistory (which is the Linked List, Array List, Hash Map that we are testing), it does this through the generateFEN() function which will be described later.



```java
} else {
    if (isValidMove(selectedRow, selectedCol, row, col)) {
        Piece movedPiece = squares.get(selectedRow).get(selectedCol).getPiece();

        if (movedPiece.getType() == PieceType.KING_WHITE || movedPiece.getType() == PieceType.KING_BLACK) {
            if (movedPiece.getType().isWhite()) {
                whiteKingMoved = true;
            } else {
                blackKingMoved = true;
            }

            if (Math.abs(selectedCol - col) == 2) {
                boolean kingside = col > selectedCol;
                int rookCol = kingside ? 7 : 0;
                int newRookCol = kingside ? col - 1 : col + 1;

                squares.get(row).get(newRookCol).setPiece(squares.get(row).get(rookCol).getPiece());
                squares.get(row).get(rookCol).setPiece(null);

                if (movedPiece.getType().isWhite()) {
                    whiteRooksMoved.set(kingside ? 1 : 0, true);
                } else {
                    blackRooksMoved.set(kingside ? 1 : 0, true);
                }
            }
        }

        if (movedPiece.getType() == PieceType.PAWN_WHITE || movedPiece.getType() == PieceType.PAWN_BLACK) {
            handlePawnMove(selectedRow, selectedCol, row, col);
        }

        squares.get(selectedRow).get(selectedCol).setPiece(null);
        squares.get(row).get(col).setPiece(selectedPiece);
```

Figure 3.1.3 *Code Snippet of actually moving the pieces.*

Another important thing would be converting the pieces within these squares to their respective letters in the Forsyth-Edward Notation (King -> K, Queen -> Q, etc.) To achieve this we made two static final Hashmaps to convert a FEN notation to a piece and vice versa as we do not want them to be edited or changed at all as we run this application, along with two functions to place each Piece to a particular FEN Character and vice versa within the hashmaps.

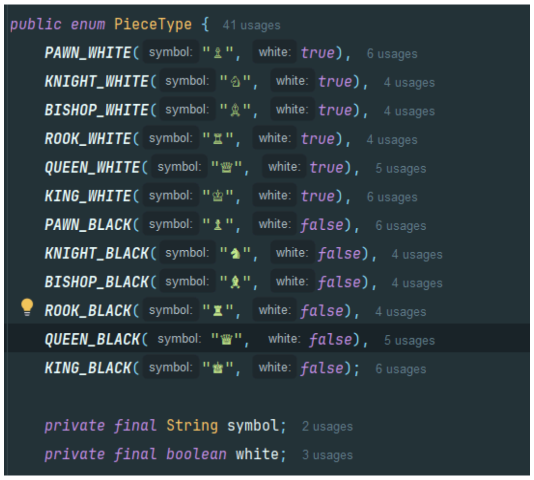


```java
private static final Map<Character, PieceType> fenToPiece = new HashMap<>();  13 usages
static {
    fenToPiece.put('P', PieceType.PAWN_WHITE);
    fenToPiece.put('N', PieceType.KNIGHT_WHITE);
    fenToPiece.put('B', PieceType.BISHOP_WHITE);
    fenToPiece.put('R', PieceType.ROOK_WHITE);
    fenToPiece.put('Q', PieceType.QUEEN_WHITE);
    fenToPiece.put('K', PieceType.KING_WHITE);
    fenToPiece.put('p', PieceType.PAWN_BLACK);
    fenToPiece.put('n', PieceType.KNIGHT_BLACK);
    fenToPiece.put('b', PieceType.BISHOP_BLACK);
    fenToPiece.put('r', PieceType.ROOK_BLACK);
    fenToPiece.put('q', PieceType.QUEEN_BLACK);
    fenToPiece.put('k', PieceType.KING_BLACK);
}

private static final Map<PieceType, Character> pieceToFen = new HashMap<>();  13 usages
static {
    pieceToFen.put(PieceType.PAWN_WHITE, 'P');
    pieceToFen.put(PieceType.KNIGHT_WHITE, 'N');
    pieceToFen.put(PieceType.BISHOP_WHITE, 'B');
    pieceToFen.put(PieceType.ROOK_WHITE, 'R');
    pieceToFen.put(PieceType.QUEEN_WHITE, 'Q');
    pieceToFen.put(PieceType.KING_WHITE, 'K');
    pieceToFen.put(PieceType.PAWN_BLACK, 'p');
    pieceToFen.put(PieceType.KNIGHT_BLACK, 'n');
    pieceToFen.put(PieceType.BISHOP_BLACK, 'b');
    pieceToFen.put(PieceType.ROOK_BLACK, 'r');
    pieceToFen.put(PieceType.QUEEN_BLACK, 'q');
    pieceToFen.put(PieceType.KING_BLACK, 'k');
}
```

Figure 3.1.4 *Conversion Functions*

As you can see above we are referring to a different class/enum called PieceType, this is another one of the classes that we have made. This class provides the particular symbol of the piece as well as having a boolean variable which dictates whether it is white or not. A lot of the functions within this class are simply getters like getSymbol() which will be used extensively when displaying the pieces on the squares to the UI.

Figure 3.1.5 *PieceType enum / class*

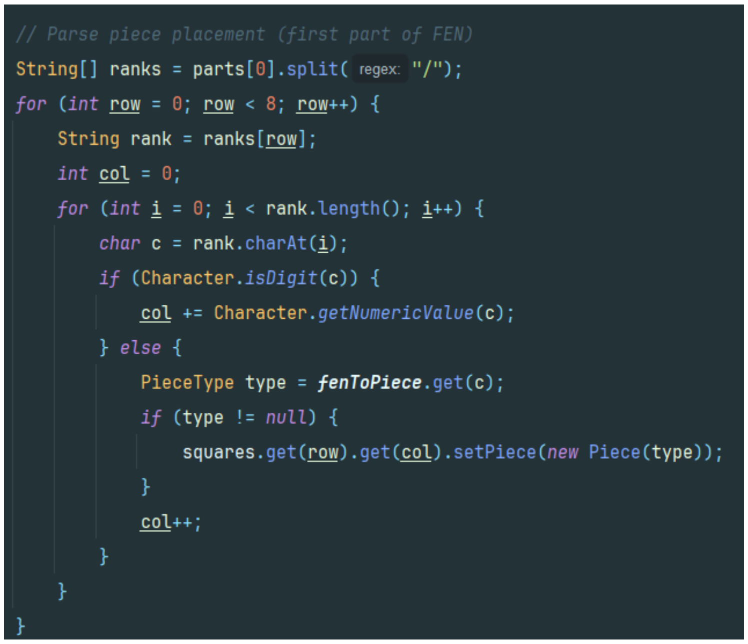


```java
public enum PieceType {  41 usages
    PAWN_WHITE( symbol: "♙", white: true),  6 usages
    KNIGHT_WHITE( symbol: "♘", white: true),  4 usages
    BISHOP_WHITE( symbol: "♗", white: true),  4 usages
    ROOK_WHITE( symbol: "♖", white: true),  4 usages
    QUEEN_WHITE( symbol: "♕", white: true),  5 usages
    KING_WHITE( symbol: "♔", white: true),  6 usages
    PAWN_BLACK( symbol: "♟", white: false),  6 usages
    KNIGHT_BLACK( symbol: "♞", white: false),  4 usages
    BISHOP_BLACK( symbol: "♝", white: false),  4 usages
    ROOK_BLACK( symbol: "♜", white: false),  4 usages
    QUEEN_BLACK( symbol: "♛", white: false),  5 usages
    KING_BLACK( symbol: "♚", white: false);  6 usages


    private final String symbol;  2 usages
    private final boolean white;  3 usages
```

➔ **FEN to Board**

Next, we will fully explain how the program actually converts a string into the squares within the chess board. The function in charge of this process is **ImportFEN().**

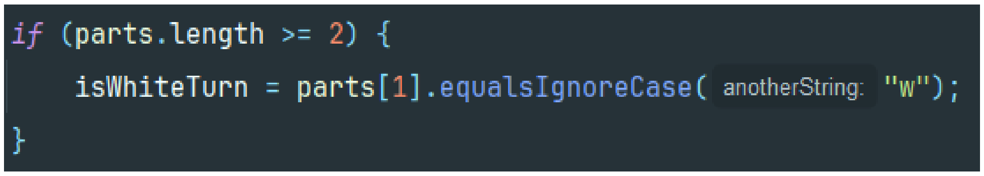
It first takes a FEN string from the fenTextField UI element then splits them by parts by space as in a FEN string the first part before the space is the location of the piece in the ranks. The first part is split once more by the slash (/) operator as that is how the notation differentiates the different ranks on the chessboard. As such there is a for loop which loops through each row then a nested for loop is used for each column then it uses a charAt() function from the string at that rank. Then it checks if there are empty squares by seeing if the number given is a digit, this is why the col is not initiated in the for loop as doing so would skip it 4 steps forward of the string and not the actual column themselves. Finally it places a piece using fenToPiece and it takes the square at that row and col and uses a setter function to place the piece.



```
// Parse piece placement (first part of FEN)
String[] ranks = parts[0].split( regex: "/");
for (int row = 0; row < 8; row++) {
    String rank = ranks[row];
    int col = 0;
    for (int i = 0; i < rank.length(); i++) {
        char c = rank.charAt(i);
        if (Character.isDigit(c)) {
            col += Character.getNumericValue(c);
        } else {
            PieceType type = fenToPiece.get(c);
            if (type != null) {
                squares.get(row).get(col).setPiece(new Piece(type));
            }
            col++;
        }
    }
}
```

Figure 3.1.6 *Import the String on the Squares*

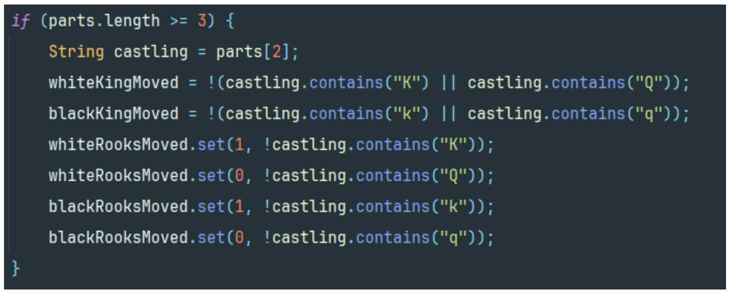
The function also checks for the current turn in the second part by checking whether there is a "w" for white or a "b" for black after the space.



```
if (parts.length >= 2) {
    isWhiteTurn = parts[1].equalsIgnoreCase( anotherString: "w");
}
```

Figure 3.1.7 *Check for the current turn*

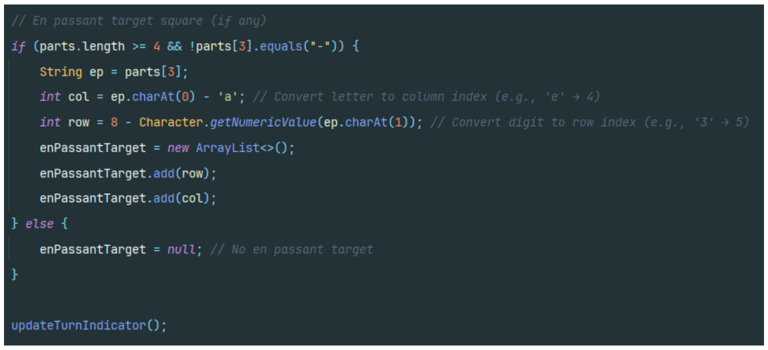
Then in the third part it checks whether castling is available by checking if the part doesn't contain either a K / k to show whether the rook has moved in the king side and a Q / q to show whether the rook has moved in the queen side, then it will update the rookMoved variables so that it can be used in the castling logic when moving a piece.



```java
if (parts.length >= 3) {
    String castling = parts[2];
    whiteKingMoved = !(castling.contains("K") || castling.contains("Q"));
    blackKingMoved = !(castling.contains("k") || castling.contains("q"));
    whiteRooksMoved.set(1, !castling.contains("K"));
    whiteRooksMoved.set(0, !castling.contains("Q"));
    blackRooksMoved.set(1, !castling.contains("k"));
    blackRooksMoved.set(0, !castling.contains("q"));
}
```

Figure 3.1.8 *Check for castling*

Finally it checks whether there is an enPassant target available, in the FEN notation this is notated by a letter then a number, with the letter starting from a representing 0 and h representing 7 to show the column and the number to show the row at which the en passant target is located, we will get into greater detail on how en passants work later when we get to the part of describing the logic of the chess pieces.



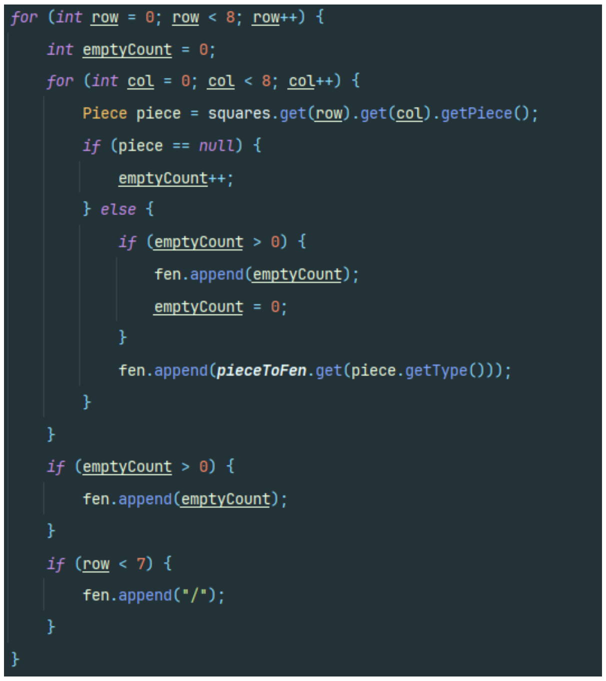
```java
// En passant target square (if any)
if (parts.length >= 4 && !parts[3].equals("-")) {
    String ep = parts[3];
    int col = ep.charAt(0) - 'a'; // Convert letter to column index (e.g., 'e' → 4)
    int row = 8 - Character.getNumericValue(ep.charAt(1)); // Convert digit to row index (e.g., '3' → 5)
    enPassantTarget = new ArrayList<>();
    enPassantTarget.add(row);
    enPassantTarget.add(col);
} else {
    enPassantTarget = null; // No en passant target
}

updateTurnIndicator();
```

Figure 3.1.9 *Check for En Passant*

➔ Board to FEN

It’s important to translate a FEN string into the chess board but it's also just as important for us to convert whatever is currently present in the board at this moment and turn it into the FEN string. This is handled by the **GenerateFEN()** function within the application.

This function begins with a StringBuilder class which allows us to more quickly modify strings in comparison to using regular strings. Then, much like before, we iterate through all the squares using a nested for loop but with an emptyCount variable as the FEN notation uses numbers to signify how many empty spaces are located within a particular row. As it iterates through the rank it uses the getPiece() function of the square to get the piece and then appends it to the Stringbuilder, if its null then the empty count will be added and if there is an empty count to begin with it will append the emptyCount first, reset it to zero, before appending the piece into the stringbuilder using pieceToFen. Then when it reaches the last row it will finally append a slash (/) to get to the next column.

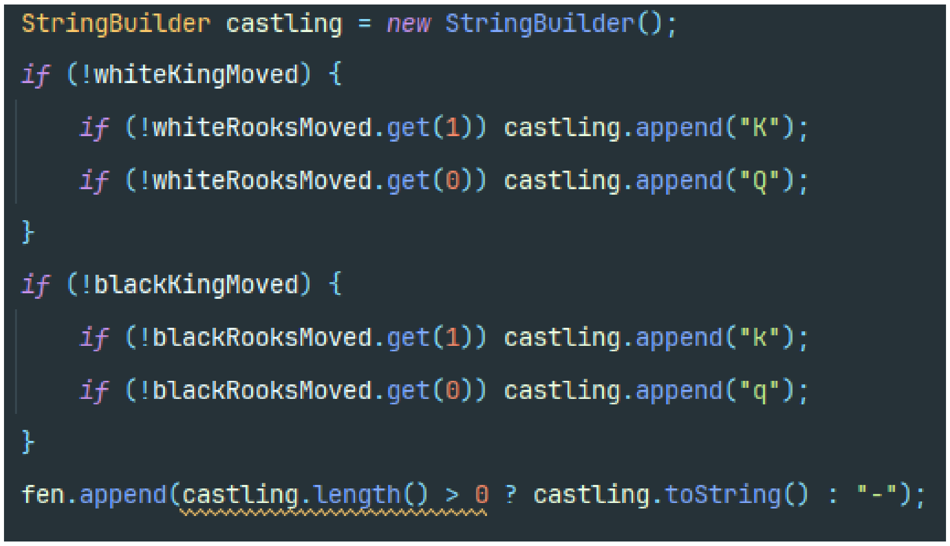


```java
for (int row = 0; row < 8; row++) {
    int emptyCount = 0;
    for (int col = 0; col < 8; col++) {
        Piece piece = squares.get(row).get(col).getPiece();
        if (piece == null) {
            emptyCount++;
        } else {
            if (emptyCount > 0) {
                fen.append(emptyCount);
                emptyCount = 0;
            }
            fen.append(pieceToFen.get(piece.getType()));
        }
    }
    if (emptyCount > 0) {
        fen.append(emptyCount);
    }
    if (row < 7) {
        fen.append("/");
    }
}
```

Figure 3.1.10 *Iterate through the ranks and squares*

Then much like the previous one it will append either "w" if isWhiteTurn is true and a "b" if isWhiteTurn is false.

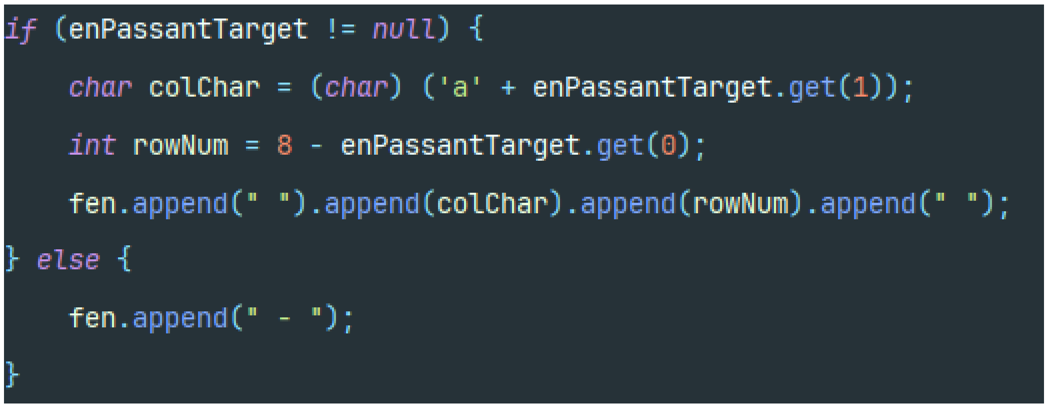
After that, it checks castling availability by first creating another Stringbuilder and looking at the whiteKingMoved, blackKingMoved, whiteRooksMoved, blackRooksMoved booleans and check one by one whether they should append K / k or Q / q then this will be appended to the FEN String if there is a length otherwise will output just a "-"



```java
StringBuilder castling = new StringBuilder();
if (!whiteKingMoved) {
    if (!whiteRooksMoved.get(1)) castling.append("K");
    if (!whiteRooksMoved.get(0)) castling.append("Q");
}
if (!blackKingMoved) {
    if (!blackRooksMoved.get(1)) castling.append("k");
    if (!blackRooksMoved.get(0)) castling.append("q");
}
fen.append(castling.length() > 0 ? castling.toString() : "-");
```

Figure 3.1.11 *Checks Castling Availability*

Like before as well, after that it will check whether there are any pieces which are deemed as an enPassant target (and once again, this shall be explained in further detail as we enter the Logic section). If the enPassantTarget is not empty then it will take the number for col and add it to the char of "a", then the row will be retrieved and minuses by 8. Finally the col is appended and the row is appended but if there is no target then it will only append a " - ".



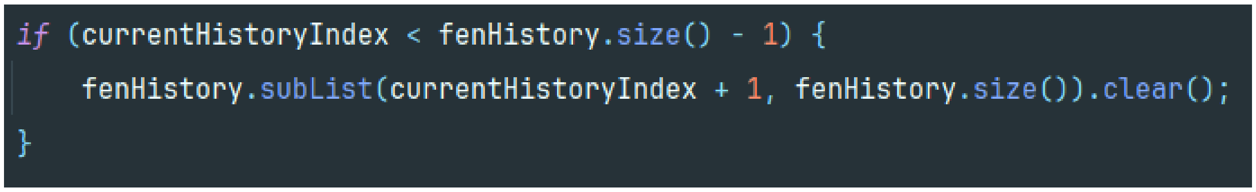
```
if (enPassantTarget != null) {
    char colChar = (char) ('a' + enPassantTarget.get(1));
    int rowNum = 8 - enPassantTarget.get(0);
    fen.append(" ").append(colChar).append(rowNum).append(" ");
} else {
    fen.append(" - ");
}
```

Figure 3.1.12 *Checks for En Passant*

Finally a FEN string usually adds the clock at the end, but since we did not implement a timer at all we only appended the basic "0 1" for all FEN strings.

➔ **Add FEN String to Move History**

Now that we know how to convert a FEN to the board and vice versa, it's important to be able to store them and string them together in a Move History (which is the data structure which we are comparing. The function in charge of this is the addToMoveHistory() function. The function first checks whether the current index is the last entry or not. If it isn't then it will choose the sublist of everything above the current index until the end and completely clear it before adding.



```
if (currentHistoryIndex < fenHistory.size() - 1) {
    fenHistory.subList(currentHistoryIndex + 1, fenHistory.size()).clear();
}
```

Figure 3.1.13 *Check if overwriting is required*

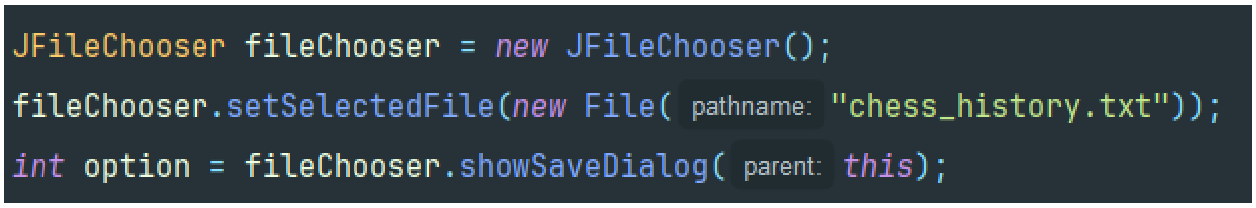
When adding a new string, it will first call the **GenerateFEN()** function which turns the current board position into a string, then it will add it into the data structure, in our example which is a linked list it will use the add() function of the linked list. After that it will update the current index to the size of the FEN history - 1 (not counting the initial FEN position). Finally, it will update the Spinner used to show the current index and navigate an index in the UI as well as updating the navigation buttons for moving back and forth.

➔ **Navigating the Move History**

This is handled with the JumpToMove() function. This function will first get the integer from the spinner then check whether this integer is larger than 0 and less than the size of the Move History. If it is, then it will set the current history index as the index provided before changing the FEN text field to the one in index, then initializing it to the board with intializeFEN()

➔ **Exporting FEN History to a File**

This feature implements JFileChooser in order to save files. Starting with exporting, this is handled by the function **exportHistoryToFile()**, the function begins with creating a new JFileChooser object then using setSelectedFile() to create a txt file called chess_history. It will then prompt the user on the location of where to save using the showSaveDialog() function.



```
JFileChooser fileChooser = new JFileChooser();
fileChooser.setSelectedFile(new File( pathname: "chess_history.txt"));
int option = fileChooser.showSaveDialog( parent: this);
```

Figure 3.1.14 *File Chooser*

It will then prompt the user on the location of where to save using the showSaveDialog() function. It will then get the path of where you want to export from the dialog into a variable before doing Files.write on the path selected. Exporting the entire list of the move history. The saved file will have all the strings be separated by a new line.

➔ **Importing FEN History to a File**

Much like the export to file function, the **importHistoryToFile()** function also utilizes JFileChooser but this time in order to read files and to place them within the current moveHistory. Much like before it starts by prompting the user with a dialog but this time it uses showOpenDialog(), this allows the application to get the path that the user specifies in a graphical manner.

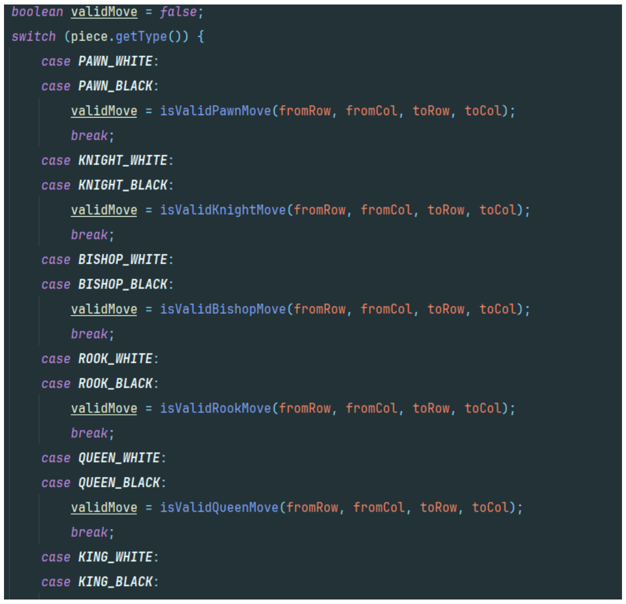
When the user selects approve then it grabs the path using getSelectedFile and toPath from the fileChooser before doing Files.readAllLines and adding each one of them into the data structure after checking that the file is not empty.

After that, it changes the current index to the very end of the moveHistory through the size of the list, changes the spinner to match, then changes the text field to show the current FEN string before using initializeFEN() to place the pieces into the board.

➔ **Checking for Move Validity**

This is now moving to the logic of the game, in order to check if a move is valid it goes through the function of **isValidMove** and **isValidAttack**. This takes 4 integers and returns a boolean, the 4 integers are fromRow, fromCol, toRow, and toCol which is taken from the HandleSquareClick function.

This function executes by first checking if the piece selected exists and is the correct color, then checks if the destination has a piece of the same color. Then it will create a boolean called validMove set to false then use a switch case to check if the move executed is according to the piece logic. If it is not a valid move it will return false.



```java
boolean validMove = false;
switch (piece.getType()) {
    case PAWN_WHITE:
    case PAWN_BLACK:
        validMove = isValidPawnMove(fromRow, fromCol, toRow, toCol);
        break;
    case KNIGHT_WHITE:
    case KNIGHT_BLACK:
        validMove = isValidKnightMove(fromRow, fromCol, toRow, toCol);
        break;
    case BISHOP_WHITE:
    case BISHOP_BLACK:
        validMove = isValidBishopMove(fromRow, fromCol, toRow, toCol);
        break;
    case ROOK_WHITE:
    case ROOK_BLACK:
        validMove = isValidRookMove(fromRow, fromCol, toRow, toCol);
        break;
    case QUEEN_WHITE:
    case QUEEN_BLACK:
        validMove = isValidQueenMove(fromRow, fromCol, toRow, toCol);
        break;
    case KING_WHITE:
    case KING_BLACK:
```

Figure 3.1.15 *IsValidMove switch case*

Then it will run a short simulation by moving the piece first without updating the UI then using the IsKingInCheck function, checks to see if the king becomes exposed, if it is then it will return false otherwise it returns true.

➔ **Pawn Movement and Attack**

isValidPawnMove() controls the logic of pawn movement, it takes the same integers as isValidMove() and the logic functions by first checking if getting the piece from the square and checking if it is white. If it is white then the direction will be negative as it is decreasing in row (black is at the top and rank 1) and black's direction will be positive as it moves downwards.

Then it checks if the column is the same and the moved square contains nothing. Then check if the row you are clicking to is the same as adding the current row + the direction, if it is then it will return true.

It also checks if the piece is within rank 6 or rank 1 (or one rank above the top and bottom edges), and it will also check if selected square is equal to 2 * the direction from the original col. If this move is done one square behind the piece will be added into the En Passant target variable.

This function also controls capture by checking if the absolute value from the subtraction of the target column (to check that they are either to the left or right) and checks if it's right in front of them. Finally it checks if the selected piece exists in that square before returning true.

It can also En Passant capture if there is an en passant target in the square selected from the previous check.

➔ **Bishop Movement and Attack**

isValidBishopMove() controls the logic of bishop movement, it takes the same integers as isValidMove() and the movement functions by insuring the value of fromRow - toRow is equal to the value of fromCol - toCol, this makes sure that the movement is diagonal as not following that will return false. After that it checks for obstructions by first getting the direction which is either positive or negative and is given by comparing if the selected row is larger than the one the piece was originally on. Next it does a while loop and checks all diagonals by adding both the column and row with the direction and checking if there is any piece blocking it that is the same color, if there is then it will return false, otherwise it will return true.

➔ **Rook Movement and Attack**

isValidBishopMove() controls the logic of rook movement, it takes the same integers as isValidMove() and the movement functions exactly the same as the bishop however instead of making sure that the value of fromRow - toRow is equal to the value of fromCol - toCol it will instead insure that the selected row is the same

➔ **Queen Movement and Attack**

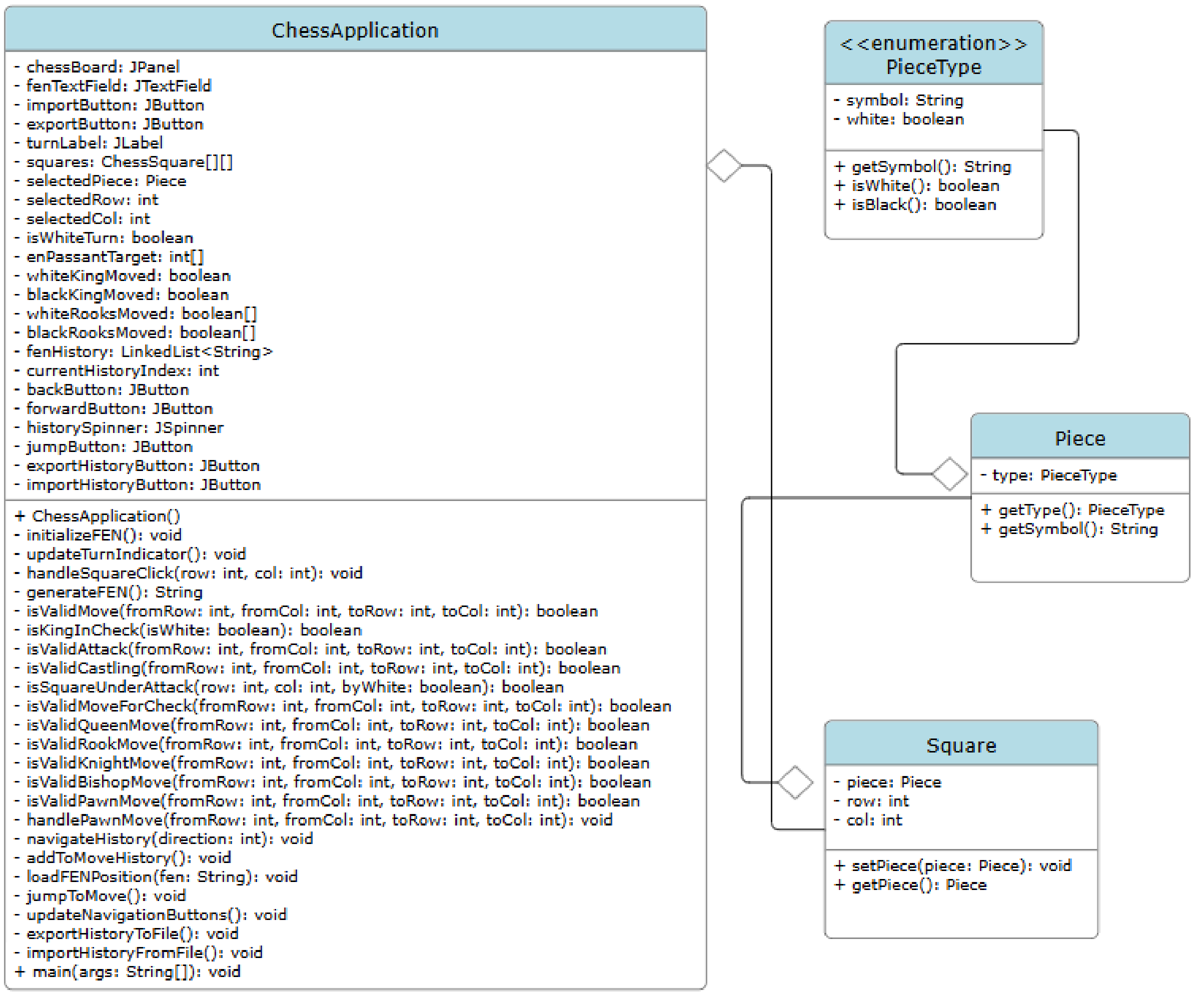
isValidQueenMove() is simply a logical or between both the bishop and rook movement as that is essentially what the queen is.

➔ **King Movement and Attack**

Moves only 1 direction around them by making sure that the absolute value of the row and column chosen and the original subtracted are less than or equal to one. However the queen has **isKinginCheck()** as a primary function to ensure that all pieces are unable to move if it leaves the king in jeopardy / check. It first checks for the king position by iterating through all the squares and finding if the pieceType in that square is a king that is the same as the current turn. Then it saves the column and row of the king.

After that it checks every row from the top to bottom, iterating with another for loop, then it checks whether any piece in any given square is able to get an attack from their row and column to the king's row and column.

3.2 Class Diagram


ChessApplication
- chessBoard: JPanel
- fenTextField: JTextField
- importButton: JButton
- exportButton: JButton
- turnLabel: JLabel
- squares: ChessSquare[][]
- selectedPiece: Piece
- selectedRow: int
- selectedCol: int
- isWhiteTurn: boolean
- enPassantTarget: int[]
- whiteKingMoved: boolean
- blackKingMoved: boolean
- whiteRooksMoved: boolean[]
- blackRooksMoved: boolean[]
- fenHistory: LinkedList<String>
- currentHistoryIndex: int
- backButton: JButton
- forwardButton: JButton
- historySpinner: JSpinner
- jumpButton: JButton
- exportHistoryButton: JButton
- importHistoryButton: JButton
+ ChessApplication()
- initializeFEN(): void
- updateTurnIndicator(): void
- handleSquareClick(row: int, col: int): void
- generateFEN(): String
- isValidMove(fromRow: int, fromCol: int, toRow: int, toCol: int): boolean
- isKingInCheck(isWhite: boolean): boolean
- isValidAttack(fromRow: int, fromCol: int, toRow: int, toCol: int): boolean
- isValidCastling(fromRow: int, fromCol: int, toRow: int, toCol: int): boolean
- isSquareUnderAttack(row: int, col: int, byWhite: boolean): boolean
- isValidMoveForCheck(fromRow: int, fromCol: int, toRow: int, toCol: int): boolean
- isValidQueenMove(fromRow: int, fromCol: int, toRow: int, toCol: int): boolean
- isValidRookMove(fromRow: int, fromCol: int, toRow: int, toCol: int): boolean
- isValidKnightMove(fromRow: int, fromCol: int, toRow: int, toCol: int): boolean
- isValidBishopMove(fromRow: int, fromCol: int, toRow: int, toCol: int): boolean
- isValidPawnMove(fromRow: int, fromCol: int, toRow: int, toCol: int): boolean
- handlePawnMove(fromRow: int, fromCol: int, toRow: int, toCol: int): void
- navigateHistory(direction: int): void
- addToMoveHistory(): void
- loadFENPosition(fen: String): void
- jumpToMove(): void
- updateNavigationButtons(): void
- exportHistoryToFile(): void
- importHistoryFromFile(): void
+ main(args: String[]): void
<<enumeration>>
PieceType
- symbol: String
- white: boolean
+ getSymbol(): String
+ isWhite(): boolean
+ isBlack(): boolean
Piece
- type: PieceType
+ getType(): PieceType
+ getSymbol(): String
Square
- piece: Piece
- row: int
- col: int
+ setPiece(piece: Piece): void
+ getPiece(): Piece

3.3 Data Structures

To obtain and evaluate the performance of the different data structures used in the chess engine (ArrayList, HashSet, and LinkedList) we performed basic operations for each engine (Moving a piece, Jump to move, importing, and exporting). The main chess engine utilizes FEN strings to represent chess positions, which allows the export and import of other chess moves via plain text files. We prepared 6 plain text files that contain different values of FEN strings from 1, 10, 100, 1000, 10000, and 100000. We then imported these files into each chess engine and recorded the performance (Time taken and Memory used) for each engine. We then tested the time and memory required to export the FEN strings of similar values. The results were then inserted into a table where the average is taken and the most suitable is highlighted (least time taken and memory used). The methods we used can be found below.

The following testing strategy was used for consistent and measurable results:

➔ **Simulating Chess Moves**

Simple chess moves were simulated (Moving the white and black queen pieces back and forth) and repeated by copy-pasting the FEN strings into plain text files until the target sizes (1, 10, 100, 1000, etc.) were reached.

➔ **Import Test**

The generated text files from the simulation were then imported using the import feature into the different chess engines to simulate the process of importing game history.

➔ **Export Test**

After importing the text files, we would then export the same text file and measure the time and memory taken for each engine.

➔ **Functionality Testing**

We also conducted simple tests to simulate more natural movements in different engines and also the time to jump to and from different moves in each engine.
"Moving a Piece" Simulates executing a move on the board.
"Jump to Move" simulates retrieving previous moves from memory.
These tests were simply done by manually moving pieces in the different chess engines.

To measure the performance of each chess engine:

➔ **Time Measurement**

To measure the time taken for each operation, Java's System.nanoTime() was used. Each result was taken from the printed values in the terminal.

➔ **Memory Usage**

The total memory consumption was recorded using runtime memory inspection (Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()). The results were measured in bytes.

a. Import Time (ms)

Operation	HashSet Time	ArrayList Time	LinkedList Time
1 Move Import	4.9361	4.9218	5.2859
10 Moves Import	5.5981	4.9077	5.4648
100 Moves Import	6.8090	6.0229	5.6820
1000 Moves Import	7.8951	7.3601	7.9610
10,000 Moves Import	15.2454	17.6329	14.6878
100,000 Moves Import	49.0184	42.0112	46.0405
Average	14.9170	13.8094	14.1870

Figure 3.3.1

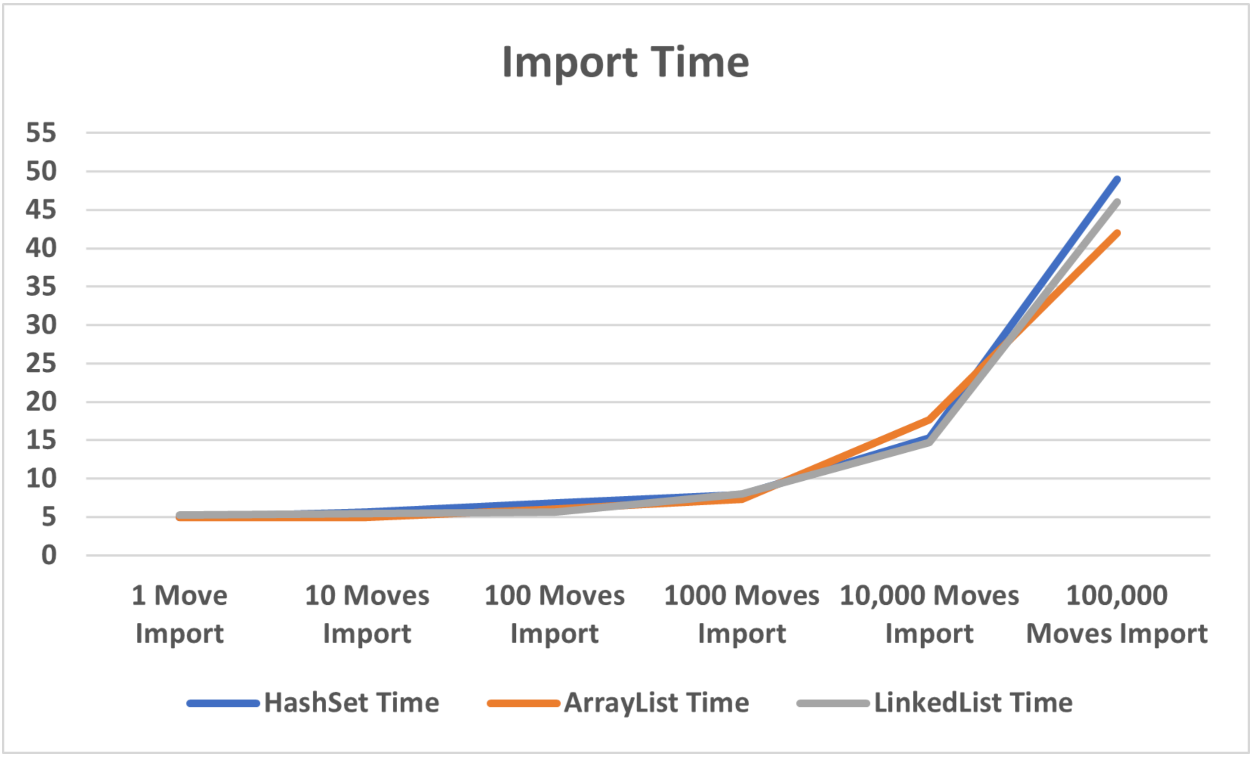


Figure 3.3.2 (Lower is better)

b. Export Time (ms)

Operation	HashSet Time	ArrayList Time	LinkedList Time
1 Move Export	1.7919	1.0047	0.9724
10 Moves Export	1.1625	0.9714	1.0002
100 Moves Export	1.4811	1.3749	1.2883
1000 Moves Export	4.0629	3.5628	3.3804
10,000 Moves Export	10.0755	6.5875	6.5328
100,000 Moves Export	31.4322	25.2073	25.7444
Average	8.3344	6.4514	6.4864

Figure 3.3.3

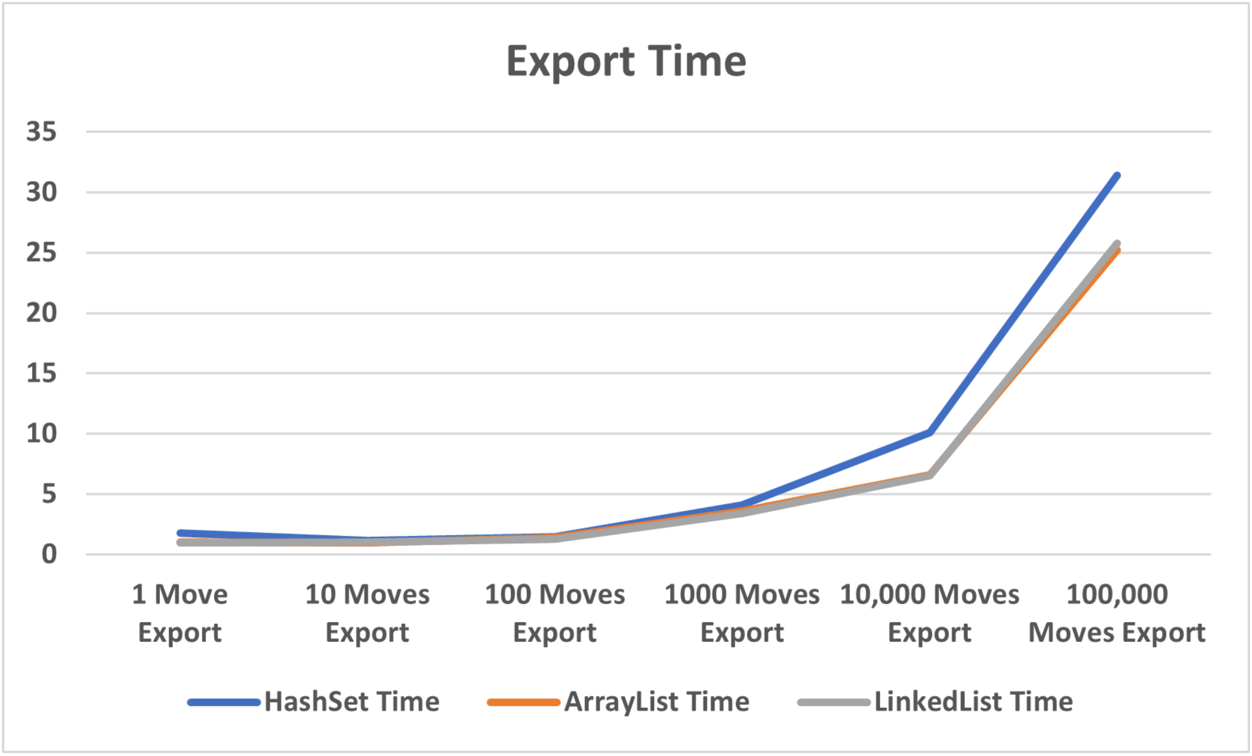


Figure 3.3.4 (Lower is better)

c. **Import Memory (bytes)**

Operation	**HashSet Memory**	**ArrayList Memory**	**LinkedList Memory**
1 Move Import	17,281,568	17,783,896	17,784,112
10 Moves Import	17,784,576	17,783,896	17,280,568
100 Moves Import	17,280,568	17,280,560	17,280,760
1000 Moves Import	17,280,600	17,783,984	17,280,568
10,000 Moves Import	18,828,296	19,848,872	20,318,136
100,000 Moves Import	25,480,240	18,132,344	21,130,200
Average	18,989,308	18,102,264	18,512,390

Figure 3.3.5

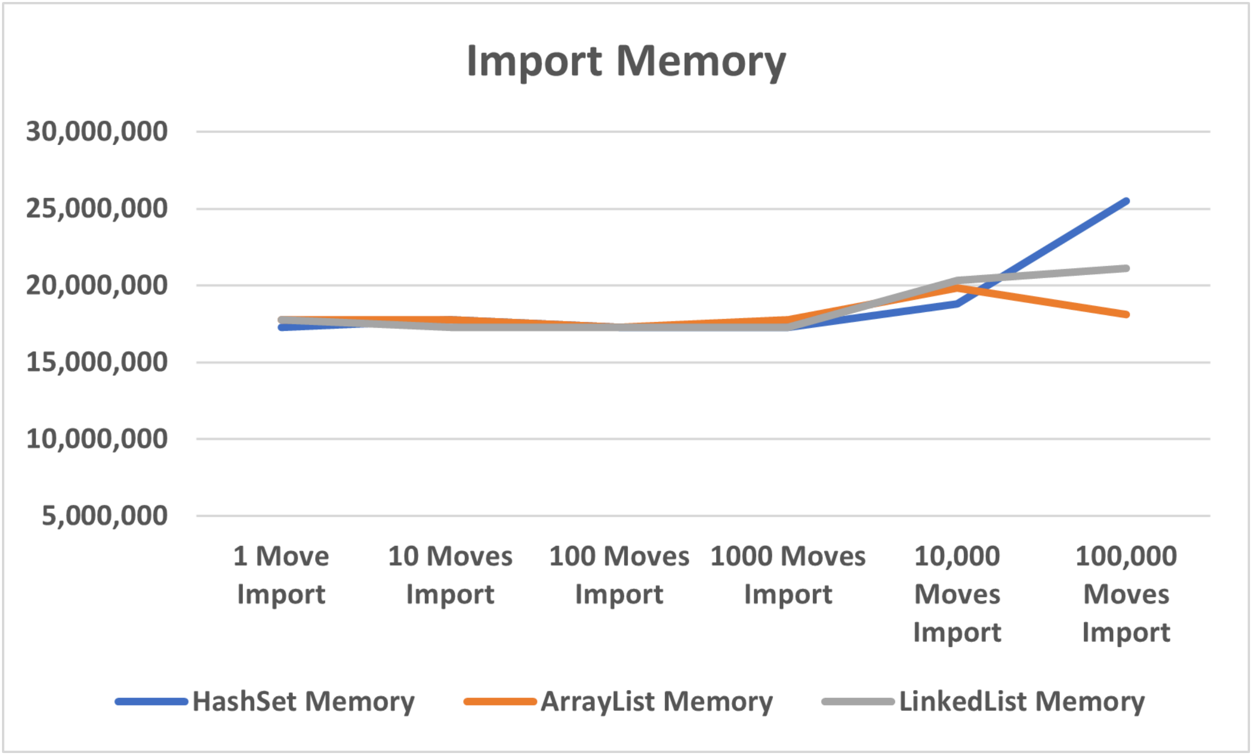


Figure 3.3.6 (Lower is better)

d. Export Memory (bytes)

Operation	HashSet Memory	ArrayList Memory	LinkedList Memory
1 Move Export	18,287,248	22,481,592	21,474,864
10 Moves Export	23,488,288	20,803,992	20,803,944
100 Moves Export	21,474,896	20,300,600	20,804,344
1000 Moves Export	21,474,872	22,984,888	20,971,520
10,000 Moves Export	25,035,952	23,991,640	23,991,824
100,000 Moves Export	31,564,744	20,397,776	23,477,528
Average	23,554,330	21,826,748	21,920,670

Figure 3.3.7

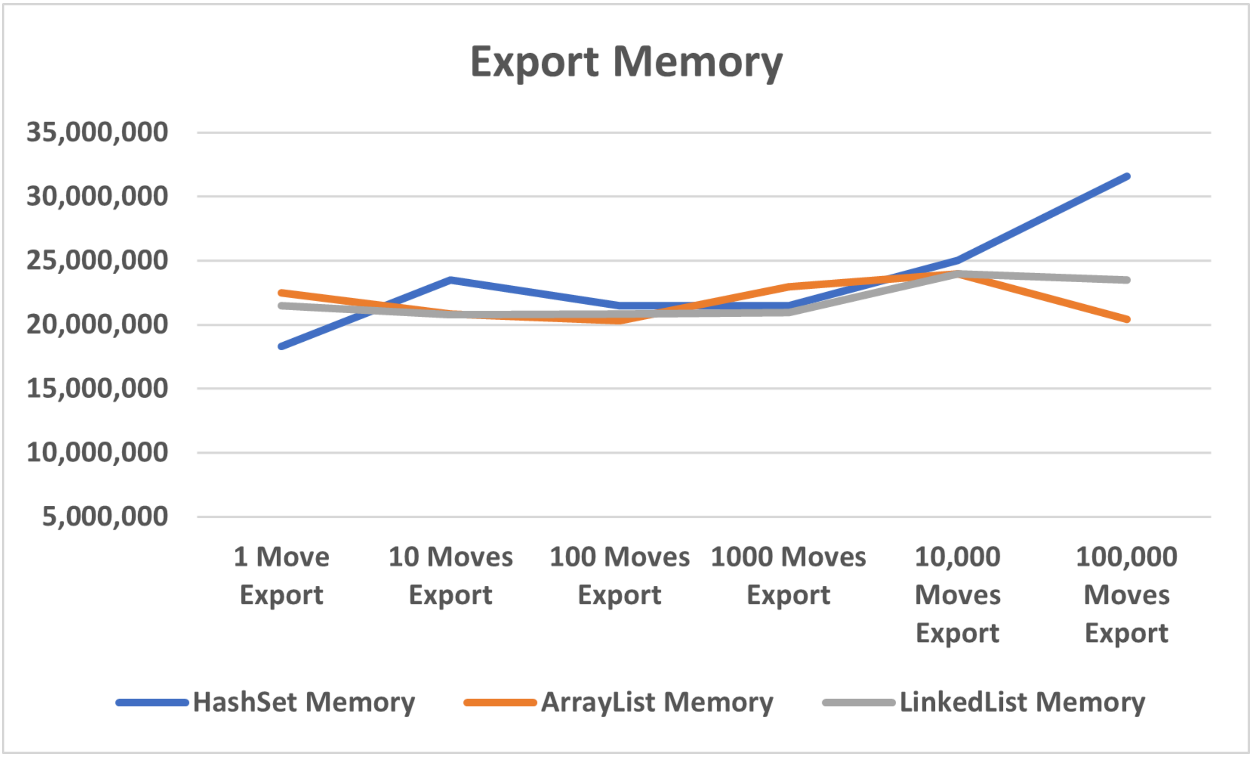


Figure 3.3.8 (Lower is better)

e. Basic Operations Time (ms)

Operation	HashSet Time	ArrayList Time	LinkedList Time
Move a Piece	2.0581	2.4090	2.7537
Jump to Move	1.9873	2.2498	2.0399

Figure 3.3.9

f. Basic Operations Memory (bytes)

Operation	HashSet Memory	ArrayList Memory	LinkedList Memory
Move a Piece	12,415,544	13,086,264	12,582,912
Jump to Move	15,099,648	15,603,016	16,106,352

Figure 3.3.10

3.4 Complexity Analysis

➔ **Import Time Export Time Taken:**

For import and Export operations, the engine that utilizes ArrayList is generally the fastest out of all the engines. This is because ArrayList allows fast appending operations, especially when the elements are added in a sequential order. Unlike LinkedList, which needs to create and link new nodes for each insertion, which requires more time. ArrayLists only need to place elements at the next index that is available in the array. HashSet is also slower as it has to check for duplicate elements and it has to calculate a hash code for each element before adding it. Importing FEN strings generally don't require these checks, so ArrayLists handle these tasks more quickly and efficiently compared to the other two engines.

➔ **Import and Export Memory Usage:**

For import and Export operations, ArrayList also uses the least amount of memory. This is due to the fact that ArrayLists stores all the data in one simple array, unlike LinkedList that requires extra memory to keep track of each element and also keep track of the links between each node. The engine that utilizes HashSet uses even more memory as it stores data in a hash table, which requires extra memory for hashing and collision handling. So, ArrayList is also the most efficient in terms of memory when importing FEN strings.

➔ **Operations (Move / Jump to Move):**

For basic operations like moving a piece or jumping to a move, HashSet is generally the fastest and most memory-efficient. This is because HashSet is designed for quick lookups by using hashing to find and store elements, making operations like checking if a move exists or switching between moves very quick. In comparison, ArrayList and LinkedList take longer to find specific items because they have to search through the list one element at a time. LinkedList also uses more memory for these operations since it has to store links between each item. So generally for fast handling of individual chess moves and searching for moves, HashSet works the best due to its efficient structure.

➔ **Definitive Best:**

As we compared the different chess engines, it is clear that the engine that utilizes ArrayList is the best option due to its overall efficiency. Although LinkedList is a valid alternative for the engine, ArrayList manages to outperform it in both time taken and memory usage during most operations. HashSet, on the other hand, is more optimized for more specific operations such as quick lookups (Jump to Move). Despite this, HashSet still consumes a significant amount of memory compared to the other two structures and it performs much slower in operations like importing and exporting large sets of data. In conclusion, ArrayList is the best option out of the three, providing a good balance between speed, memory usage, and its simplicity, making it the most practical and efficient data structure for the chess engine.

4. Team Workload

4.1 Code Stuff

- ➔ Main Chess Engine: Rafie
- ➔ ArrayList Implementation: Rafael
- ➔ LinkedList Implementation: Rafie
- ➔ HashSet Implementation: Thristan
- ➔ Rechecking / Polishing: Rafie
- ➔ Testing: Thristan
- ➔ Conversion to JAR & EXE: Rafael

4.2 Documentation

- ➔ Research / Referencing: Rafael
- ➔ Background: Rafael
- ➔ Problem Description: Rafie
- ➔ Function Description: Rafie
- ➔ Data Structures: Thristan
- ➔ Complexity Analysis: Thristan
- ➔ Appendix: Rafael
- ➔ Program Manual: Rafael

5. References

[1] Smith, J. R. (2022). *The Transformative Impact of Artificial Intelligence on Modern Society*. Journal of Emerging Technologies, 15(3), 201-215. (Previously [1])

[2] Chess Power. (n.d.). *Benefits of Playing Chess*. Retrieved from https://www.chesspower.co.nz/benefits-of-playing-chess (Placeholder for user's (Chess Power, n.d.))

[3] Frank, A. (1973-74). *Chess and Mathematical Ability*. Journal of Psychology, 89(1), 19-24. (Placeholder for user's (Frank, 1973-74))

[4] De Groot, A. D. (1974-76). *Thought and Choice in Chess*. Mouton. (Placeholder for user's (De Groot, 1974-76))

[5] Fung, K. M. (1977-79). *The Effects of Chess on the Academic Achievement of Primary School Students*. Education Journal (Hong Kong), 7(1), 7-15. (Placeholder for user's (Fung, 1977-79))

[6] Pennsylvania Department of Education. (1979-83). *Chess in Education: A Study of the Effects of Chess Instruction on Students' Cognitive Development*. Harrisburg, PA: Pennsylvania Department of Education. (Placeholder for user's (Pennsylvania Department of Education, 1979-83))

[7] ADHD & Chess. (2020). *How Chess Helps ADHD: Improving Focus and Attention*. Retrieved from https://www.adhdandchess.org/how-chess-helps-adhd (Placeholder for user's (ADHD & Chess, 2020))

[8] European Proceedings. (2019). *Computational Thinking and Chess: A Case Study*. European Proceedings of Social and Behavioural Sciences, 67, 123-135. (Placeholder for user's (European Proceedings, 2019))

[9] Children's School. (2022). *AI-Assisted Chess Training for Enhanced Learning*. Retrieved from https://www.childrensschool.org/ai-chess-training (Placeholder for user's (Children’s School, 2022))

[10] Chess.com. (n.d.). *FEN: Forsyth-Edwards Notation*. Retrieved from https://www.chess.com/terms/fen-chess (This would be a real web source if you wanted to cite it properly, but for the example, a placeholder is fine). (Previously [6])

[11] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press. (Previously [7])

[12] Wirth, N. (1976). *Algorithms + Data Structures = Programs*. Prentice-Hall. (Previously [8])

[13] Rahman, M. M., & Haque, M. R. (2021). *Performance Analysis of Online Learning Management Systems Under High Load Conditions*. Journal of Educational Technology & Society, 24(2), 112-125. (Previously [9])

[14] Liu, Y., & Chen, G. (2019). *Optimizing Game Tree Search through Efficient Data Structures for State Representation*. Proceedings of the International Conference on Game Development, 12, 89-102. (Previously [10])

[15] Knuth, D. E. (1997). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley. (Previously [11])

6. Appendix

A. Main Dependencies For The Program

- Latest Java Version: https://www.oracle.com/id/java/technologies/downloads/#jdk24-windows

B. Program Manual, How To Run Program (With Screenshots)

- Program Manual: https://docs.google.com/document/d/1EY0CoVqfYtyWZxzc_QBwI8KWDcFgr8UBBPjxLdm3XPo/edit?usp=sharing

C. Link To GitHub Repo

- GitHub: https://github.com/rafipy/chess-engine-FEN/tree/main

D. Link To Poster

- Poster: https://www.canva.com/design/DAGqWJMEm1M/DKi2_uOWWECX8s1rINvqcw/edit