Final Report Data Structure

**Tic-Tac-Toe:**

The game code is a Java implementation of the classic game Tic-Tac-Toe. It uses a HashMap to represent the game board, allowing players to take turns placing their marks ('X' or 'O') on the board until a winner is determined or the game ends in a draw.

# Data Structures:

The primary data Structure used in the game is HashMap named Board. This HashMap

represents the game board, where the keys are integers representing the positions on the board ranging from 1 to 9, and the values are characters such as \_, X, O, representing the state of each position. Apart from HashMap, the code also uses other built-in data structures provided by java, such as Scanner for user input and String for storing user choices and values.

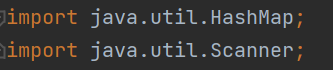
# HashMap:

In data structures, a HashMap is an implementation of an associative array or dictionary that stores key-value pairs. It is a fundamental data structure that provides efficient storage,

retrieval, and manipulation of data based on keys.

A HashMap is designed to achieve fast access to elements by using a technique called hashing. It uses a hash function to convert keys into an index within an underlying array. This index determines the position where the corresponding value is stored. By using this index-based approach, HashMap achieves constant-time average-case complexity for basic operations like

insertion, retrieval, and deletion.



## Advantages:

1. One of the key advantages of HashMap is its ability to handle large amounts of data

efficiently. It dynamically resizes itself as the number of elements increases or decreases, ensuring optimal memory usage. This dynamic resizing mechanism allows HashMap to adapt to changing data sizes without a significant performance impact.

1. HashMap also provides flexibility in terms of the types of keys and values it can store. It supports a wide range of data types, including primitive types and objects. Moreover,

HashMap allows null values and a single null key, meaning you can store a null value associated with a specific key and have a key with a null value.

1. Handling collisions is an important aspect of HashMap. Since different keys can

potentially hash to the same index, collisions occur. HashMap employs various collision resolution techniques to address this issue. The most used technique is separate

chaining, where each index in the underlying array contains a linked list of key-value pairs that share the same hash code.

**Analyses:**

## Code with HashMap Data Structure:

In the code provided, the use of a HashMap “Board” allows for a concise representation of the game board, where the keys correspond to the positions on the board (1 to 9), and the values represent the state of each position ('\_', 'X', or 'O'). This HashMap simplifies the operations of filling the board, checking for a winner, and checking for a draw. The board positions can be efficiently accessed and updated using the keys, and the code can easily determine the state of a row, column, or diagonal by checking the corresponding positions in the HashMap.

### Complexity:

Overall, the time complexity of the code written is 0(1) as the size of board is fixed and the number of interactions and operations are constant.

Where the space complexity of the code overall is 0(1) as it does not depend on the input size but on the fixed size data structure and variables used.

# Init Board:

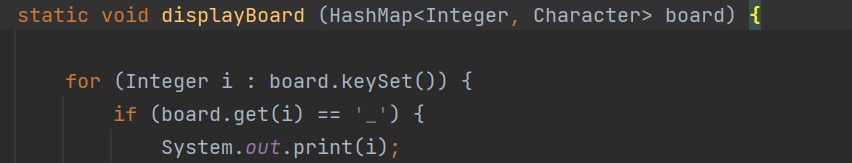
Let's start by examining the structure and purpose of each function in the code. The Init Board function initializes the game board by populating the HashMap with nine key-value pairs. The keys represent the positions on the board (from 1 to 9), and the initial value for each position is set to '\_'. This function is called once at the beginning of the game.

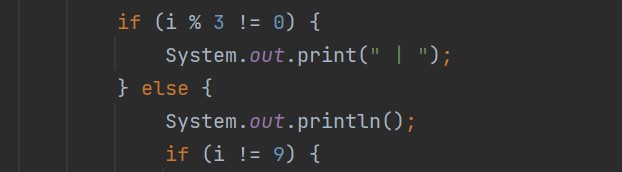


# Display Board:

The display Board function is responsible for printing the current state of the game board to the console. It iterates through the HashMap using a for-each loop and prints the value at each position. If the value is '\_', it prints the corresponding key (position number). Otherwise, it

prints the value itself. The function also handles the formatting of the board by adding '|' separators between positions and displaying horizontal lines to separate rows.





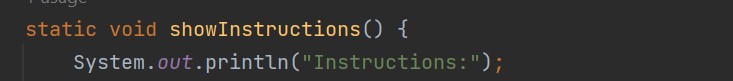
# Greet Function:

The Greet function simply prints a welcome message to the console.



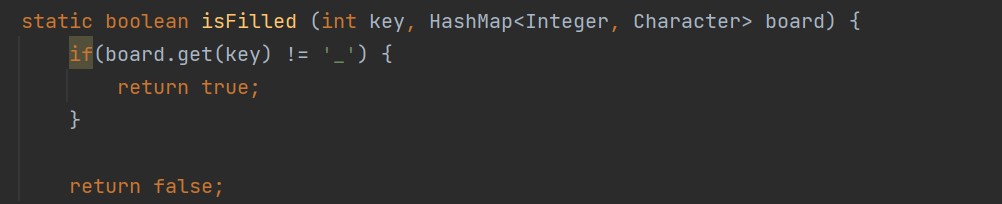
# Show Instructions:

The Show Instructions function provides the players with game instructions. It prints a set of rules and guidelines for playing Tic-Tac-Toe.



# Is Filled Function:

The Is Filled function checks if a given position on the board is already filled (i.e., not equal to '\_'). It takes the key (position) and the board HashMap as parameters and returns true if the position is filled and false otherwise.



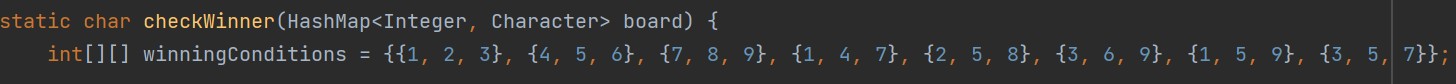
# Fill Board:

The Fill Board function updates the board by placing a player's mark (character) at a specified position. It takes the position, the board HashMap, and the mark (element) as parameters. The function replaces the value at the specified position with the mark.



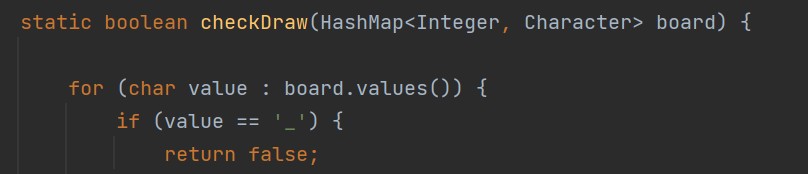
# Check Winner:

The Check Winner function determines if there is a winner in the current state of the game. It checks all possible winning combinations (rows, columns, and diagonals) by iterating over an array of winning conditions. If any winning condition is met, the function returns the winning mark ('X' or 'O'). Otherwise, it returns '\_'.



# Check Draw:

The Check Draw function checks if the game has ended in a draw. It iterates over the values of the board HashMap and returns true if there are no more empty positions ('\_') on the board, indicating a draw. Otherwise, it turns out false.

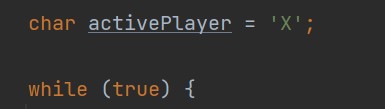


# Main Function:

The Main function is the entry point of the program. It initializes the board, displays the initial state of the board, and greets the players. It then asks if the players need instructions and

displays them if requested. The function sets the active player to 'X' and enters a while loop that continues until the game is over.

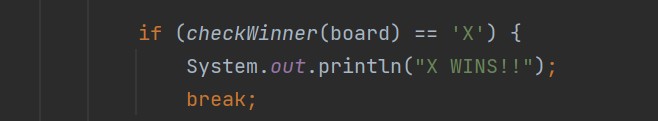


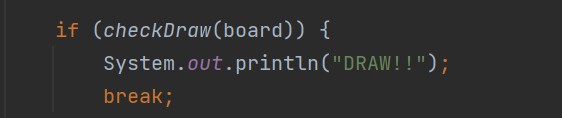


Within the loop, the function prompts the active player to enter the position where they want to place their mark. It validates the input, checking if the position is within the valid range (1 to 9) and if it is already filled. If the input is valid, the function updates the board by placing the mark at the specified position and switches the active player.

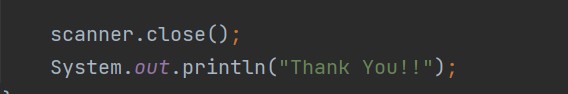


After each move, the function displays the updated board. It then checks if there is a winner by calling the Check Winner function. If a winner is found, it prints the corresponding message and breaks out of the loop. If no winner is found, the function checks for a draw by calling the Check Draw Function. If a draw is detected, it prints the draw message and breaks out of the loop.





Once the loop is exited, the function closes the scanner and prints a thank you message.



# Conclusion:

The game is a basic implementation of Tic-Tac-Toe using a HashMap as the data structure for the game board. It handles the game logic, input validation, and result determination. In which two players can play and enjoy the simple game. The code using data structure such as HashMap is easy to understand and easily readable as compared to the code written in without using data structure. The simple implementation of Tik-Tac-Toe using data structure HashMap.

# Simple code using Array:

The code is an implementation of a Tic-Tac-Toe game using a console interface. The game is designed to be played by two players, 'X' and 'O', who take turns marking cells on a 3x3 grid. The code utilizes a 2D array called "board" to represent the game board.

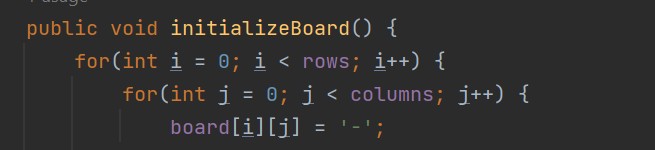
# Tic-Tac-Toe:

The Tic-Tac-Toe class serves as the main class that handles the game logic. It contains methods to initialize the board, print the current state of the board, check for a winner, check for a draw, and update the board with player moves. The class also keeps track of the current player using the current player variable.



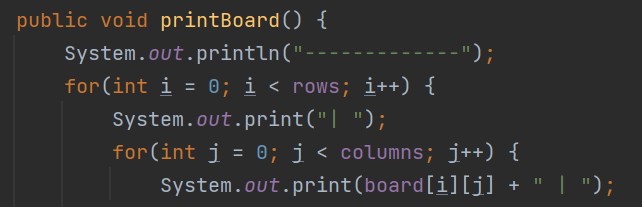
# Initialize Board:

The initialize Board method sets all the cells of the board to the empty value '-'. This method is called at the beginning of the game and whenever the game needs to be reset.



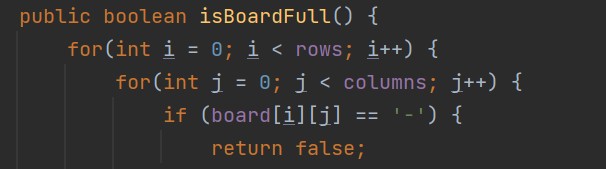
# Print Board:

The Print Board method displays the current state of the board on the console. It iterates over the rows and columns of the board and prints the corresponding cell values, separated by '|' and '-' characters to create a grid-like structure.



# Is Board Full:

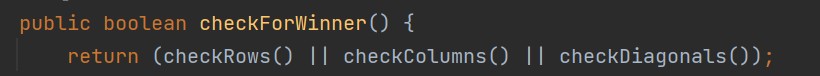
The Is Board Full method checks whether the board is filled with player marks. It iterates over all cells of the board and returns false if it finds any empty cell ('-'), indicating that the board is not full. Otherwise, it returns true.



# Check For Winner:

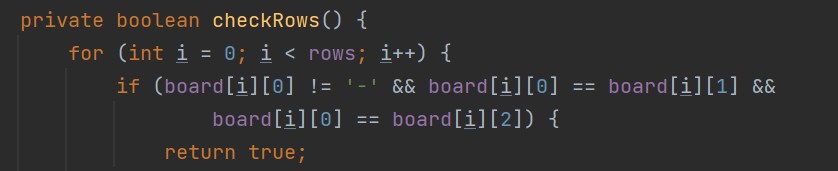
The Check for Winner method checks for a winning condition on the board. It calls three separate helper methods: check rows, check columns, and check diagonals. Each of these

methods examine the rows, columns, and diagonals of the board, respectively, to see if any player has filled a complete line with their marks. If a winning condition is found, the method returns true; otherwise, it returns false.



# Check Rows:

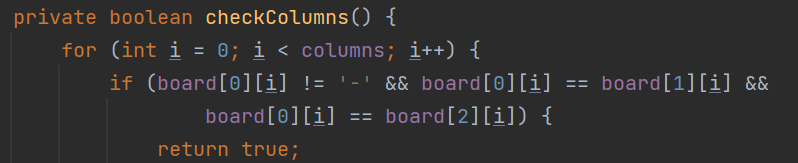
The Check Rows method iterates over the rows of the board and checks if any row has all the cells filled with the same player mark ('X' or 'O'). If a winning row is found, the method returns true; otherwise, it continues checking other rows and eventually returns false.



# Check Columns:

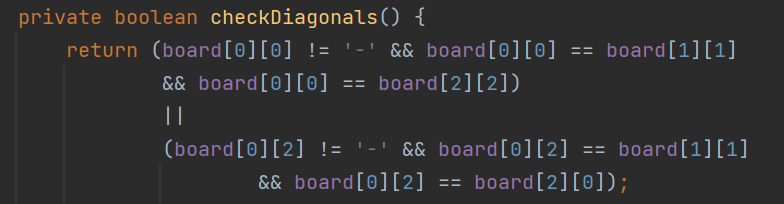
Similarly, the Check columns method iterates over the columns of the board and checks if any column has all the cells filled with the same player mark. If a winning column is found, the

method returns true; otherwise, it returns false.



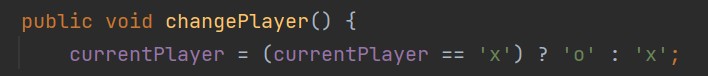
# Check Diagonals:

The Check diagonals method checks the two diagonals of the board to see if either diagonal has all the cells filled with the same player mark. If a winning diagonal is found, the method returns true; otherwise, it returns false.



# Change Player:

The Change player method updates the current player variable to switch the active player between 'X' and 'O'. It uses a ternary operator to toggle between the two players.



# Update Board:

The Update Board method is responsible for getting the player's move and updating the board accordingly. It prompts the current player to enter their move by specifying the row and column numbers. It checks if the input coordinates are valid (within the board's dimensions and the selected cell is empty) and updates the board with the player's mark if the input is valid. If the

input is invalid, the method displays an error message and prompts the player to try again.

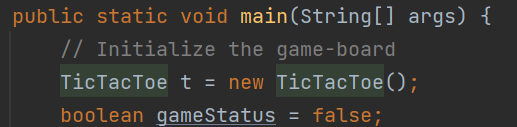


# Main:

The Main method serves as the entry point of the program. It creates an instance of the Tic-Tac-

Toe class, initializes the game board, and starts a loop that continues until a winner is determined or a draw occurs. Inside the loop, it calls the update board method to get the

player's move, prints the current state of the board, and checks if a winning condition or draw has been reached. If the game is over, it displays an appropriate message and exits the loop.



# Conclusion:

The code implements a console-based Tic-Tac-Toe game using a 2D array to represent the board. It allows two players to take turns marking cells and checks for winning conditions and draws. The implementation of the game using 2D array is slow and has different time

complexity.

# Analysis of 2D array code:

## Code with 2D array:

On the other hand, a Tic-Tac-Toe game code without a HashMap would typically use a different data structure to represent the board, such as a multidimensional array or a simple one-

dimensional array. In this case, the code would need to manually track the positions and states of the board elements using array indices. The logic for checking the board and updating the positions would be more complex and involve additional loops or conditionals to navigate the array and perform the necessary operations.

### Complexity:

It has the same time complexity will be o(n) for single array and O(n^n) if this multi-dimensional and space complexity 0(1) as the size on the board will be same and fixed and as space

complexity will depend on the fixed size.

**LinkedList:**

A LinkedList is a linear data structure that consists of nodes with each of its nodes having two fields or two heads with one head leading to the actual data and the other head leads to the next node.

A LinkedList is a also a dynamic data structure. Therefore, the size of the data can grow and shrink as much as it is needed by allocating and deallocating memory which also makes insertion and deletion easier than other data types for there is no need to shift or re shift the elements to a specific order after the said actions.

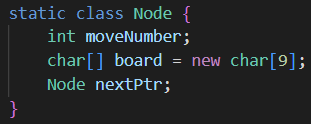
**Advantages:**

1. It’s a more dynamic data structure
2. No memory wastage due to the nature of the list being increased or decreased according to the run time without having a predestined size.
3. Easier implementation with easy insertion and deletion
4. Efficient for larger data size due to its dynamic attitude
5. Flexible and easier to scale due to how easy it is to insert and remove data at any given position inside the list

**Analyses:**

**LinkedList Game Code:**

With LinkedList, we can create a tic-tac-toe game that stores and display all the moves and the previous moves that has been played. Such things are possible for with LinkedList, the input of the two players is counted as integers and those integers become index value that represents each cell or square just like in a normal tic-tac-toe game. The integers that were given by the two players are then stored and recorded to create a history of the moves that has been played.



**Complexity:**

The time complexity for using LinkedList as a tic-tac-toe base is O(1) for the size of the board is fixed and no deletion is required. Insertion is only done towards the head therefore making it O(1).

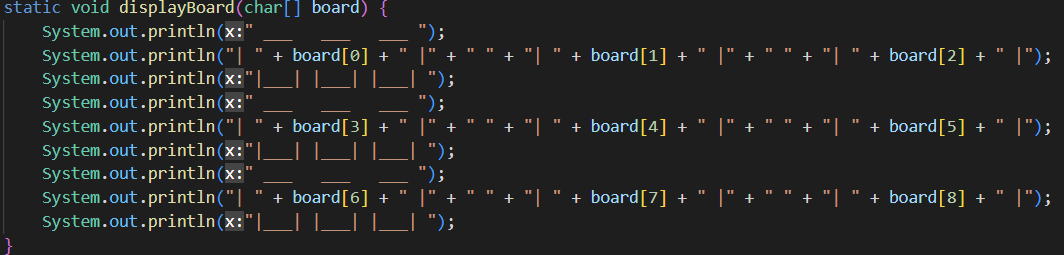
The space complexity of this algorithm is considered to be O(1) as well as no extra space is required for any operation and due to the board having a fixed size.

**Initialize Board:**

This method is used to start the game by setting the board cells into empty values for the player to input their inputs.

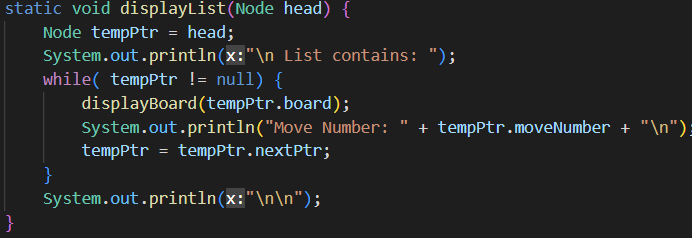


**DisplayBoard**:

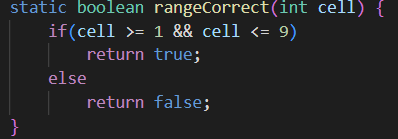
This method is used to print the board for the players to play.

**DisplayList**:

This method is used to display the previous moves.

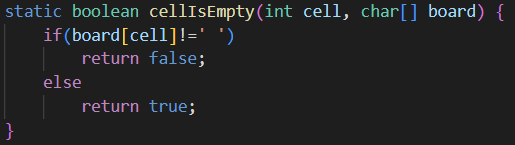


**RangeIsCorrect:**

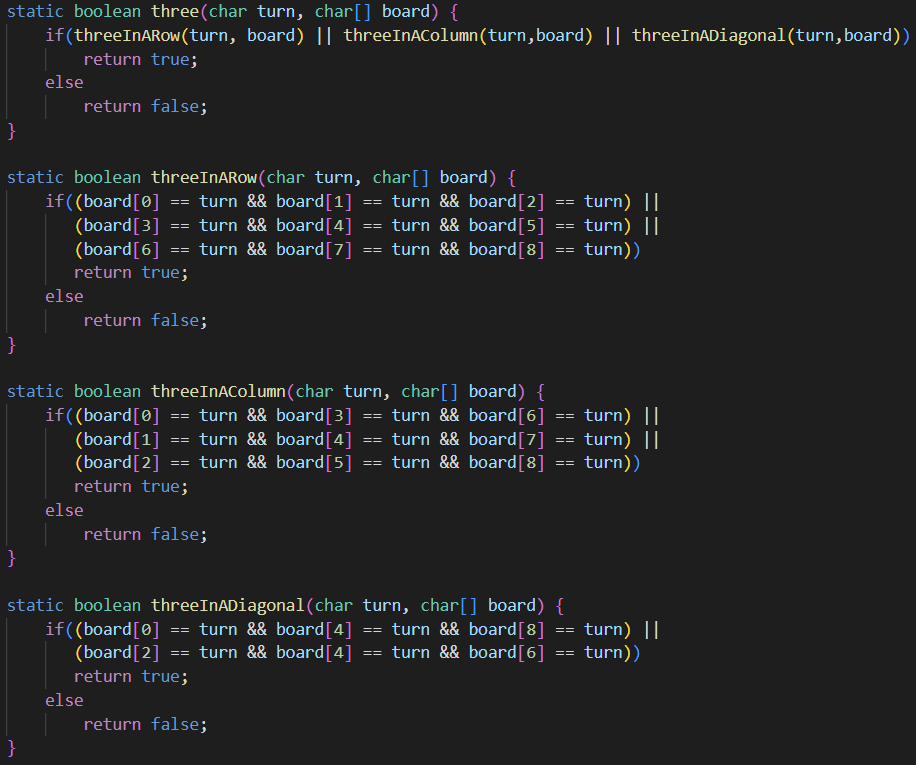
This method is used to check if the input that was given by the players are correct or not, in between 1 to 9.

**CellIsEmpty**:

This method is used to check if the cell or the square that the player has chosen is an empty cell or if it has already been taken by the other player.



**Three:**

This method or function is used to check the winner of the game and to check if a three in a row, horizontal, or diagonally has been made that could be the cause of the winning game.

**Conclusion:**

LinkedList is used as the base for this game and it allows this version of the game to store all the previous moves that has been played by the players.

**Benchmark:**

To see which data structure is faster when it comes to building the tic-tac-toe game, we tested the game ten times with each data structures and with a fixed draw position.

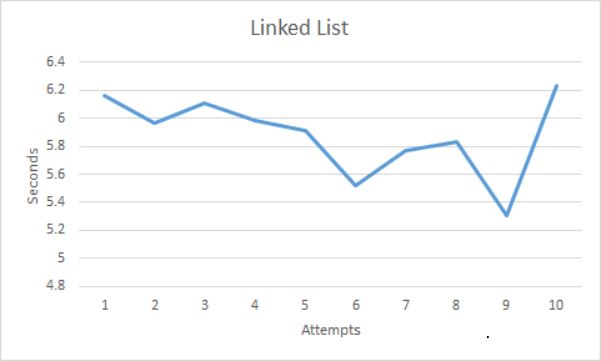
Each data structure will be tested using the same device and with the same method. The device that will be used to test out all the different data structure game is as follows:

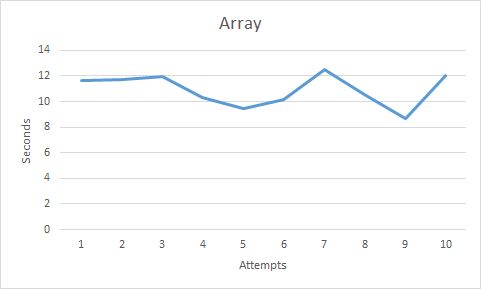
* Processor :   
  11th Gen Intel(R) Core(TM) i7-11370H @ 3.30GHz, 3302 Mhz, 4 Core(s), 8 Logical Processor(s)
* Memory :   
  Physical Memory (RAM) 16.0 GB

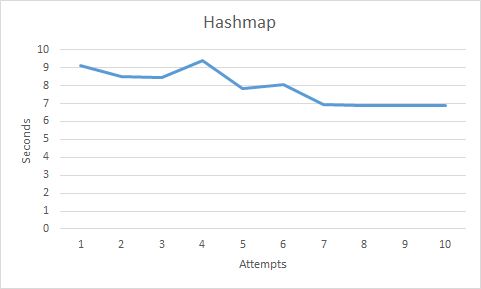
After the said tests, all the time has been collected and here are the results:

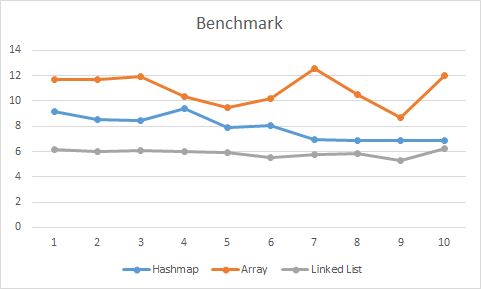
|  |  |  |
| --- | --- | --- |
| HashMap | Array | LinkedList |
| 9.12 secs | 11.66 secs | 6.16 secs |
| 8.53 secs | 11.68 secs | 5.97 secs |
| 8.46 secs | 11.94 secs | 6.11 secs |
| 9.38 secs | 10.31 secs | 5.98 secs |
| 7.86 secs | 9.45 secs | 5.91 secs |
| 8.08 secs | 10.17 secs | 5.52 secs |
| 6.96 secs | 12.52 secs | 5.77 secs |
| 6.90 secs | 10.54 secs | 5.83 secs |
| 6.90 secs | 8.66 secs | 5.31 secs |
| 6.88 secs | 12.02 secs | 6.23 secs |

Based on the times that have been noted, here are some charts for each data structures alongside a chart that compares all three data structures.









**Conclusion:**

After all the benchmarks were done, it has been proven by time that the LinkedList module seems to be working faster compared to both the HashMap module and Array module. Despite being faster at the start of the tests, the LinkedList module does ended up being similar to the HashMap module. This is due to the similar controls and inputs for each data structure methods.

However, the LinkedList module seems to be more difficult and harder to use due to its dynamic nature and how the list of the moves needs to be displayed over and over after every single move. The module can be designed otherwise but if it were to be designed as so, there would be no need for a LinkedList base in the first place. The LinkedList module also uses more space and memory compared to Array and HashMap due to the pointer that is present with the LinkedList module.

Therefore, it would be better to use HashMap as the main base foundation to create a simple tic-tac-toe game in Java with the concept of data structures.