TARLAC STATE UNIVERSITY
COLLEGE OF COMPUTER STUDIES
OPERATING SYSTEMS - OS
2nd Semester – A.Y. 2024-2025

Simulation and Analysis of a Page-Replacement Algorithm in a Virtual Memory System

A Project

Submitted to the Faculty of the College of Computer Studies
San Isidro Campus, Tarlac City,

Tarlac State University

In Partial Fulfillment of the Requirements for the Subject

Operating Systems

Academic Year 2024-2025

Submitted By:
Cervantes, Elijah Gabe C.

Arielle Joy I. Barcelona
Subject Instructor

Date of submission: May 21, 2025

Table of Contents

Project Overview

This case study aims to demonstrate how a page-replacement algorithm can be implemented (in real world applications) to a specific programming language and virtual machine. Simultaneously conducting an in-depth analysis within the algorithm itself. The purpose of this case study is to investigate how the algorithm works, assess its efficiency, and evaluate its specific outcomes to gain understanding or comprehension within the algorithm's complexity and learn about the algorithm itself.

After careful consideration, the developer decided to use FIFO as the algorithm that will be implemented to a programming language and virtual machine. FIFO is a page replacement algorithm that selects the page first entered within the memory and replace it with a new requested page if a specific page is not present within the main memory. In other words, if a page fault occurs, FIFO replaces the oldest page with a new one [1]. To further elaborate, a virtual memory system needs an efficient page-replacement algorithm to decide which page is needed to remove within the main memory in case of a page fault [2]. In other words, the page-replacement algorithm enacts as the "judge" specifically for removing a page within the main memory.

To solidify its implementation and achieve simulation, the algorithm needed a programming language. The developer has a resolution to utilize Java™ programming language – a general purpose, class-based, strongly typed and high-level programming language) [3]. However, there is a question lurking in the background, how the FIFO algorithm is implemented within the programming language? Providing a brief context, the developer utilized the following classes within Java™.

Arrays (Figure 3 and 20 – Code Explanation) - (used to store multiple values within a single variable and it's defined with square brackets) [4]. Furthermore, array is used to display the reference string of the program (simulation).

Queue (Figure 4 – Code Explanation) - (an interface that stores and processes data in order, the elements are inserted in the rear (end) and removed from the front) [5]. Queue is used to simulate the process of the page replacement algorithm since queue implements the idea of First-In, First-Out.

Another Java™ class is also utilized by the developer. **Random** (Figure 15 and 21 – Code Explanation) - (a thread-safe class used to generate pseudorandom numbers and provides various calls to generate multiple data types such as int, double and float) [6]. From the word itself "Random", this class is used to generate the reference string with 20 random page numbers (initial requirement) within the program (simulation) itself. With the help of Random class, the developer's work became much easier and lighter since the program itself automatically generated the page numbers of the reference string.

Code Explanation

Explanation of the program's source code is an essential task for this documentation. It provides a contextual understanding of the program's logic and simultaneously explaining the algorithm's implementation to the mentioned programming language. Here is the breakdown and explanation for each block of code:

```
1  //Importing the essential Java libraries for the program.
2  import java.util.Arrays;//For displaying the reference string
3  import java.util.Queue;//For the queue that will be utilized by the algorithm
4  import java.util.LinkedList;//Specialized type of queue utilized in the program
5  import java.text.DecimalFormat;//For Hit Ratio and Page-Fault Ratio
6  import java.util.Scanner;//For user prompt specifically for a number (Frame size)
7  import java.util.Random;//For generating a random string of numbers
8
```

Figure 1 – Imported Java Libraries

Importing Java Libraries is a crucial part for the simulation of the aforementioned algorithm. These java libraries provided methods can be managed and utilized within the program. Arrays, queue and random classes functionality are already mentioned in the previous chapter (Project Overview). Additional libraries' purpose and functionality such as *DecimalFormat* (Figure 2), *LinkedList* (Figure 4) and *Scanner* Classes (Figure 15 and 17) are to implement the program's additional features.

```
9  // The Java class for the page-replacement algorithm.
10 class FIFO_algorithm{
11     //Declaration of the decimal format class for the Hit Ratio and Page Fault Ratio.
12     private static final DecimalFormat df = new DecimalFormat( pattern: "0.000");
13
```

Figure 2 – Declaration of class FIFO_algorithm and DecimalFormat

FIFO_algorithm class will serve as a template for the object instantiated in the program's main method. This class functionality is to compress the algorithm's logic into a one coherent method called implementation (Figure 3). Another purpose of this class is to elevate the program's complexity. It is true that implementing the algorithm itself in the main method is more efficient. The developer wanted an elegant and eloquent way to implement the algorithm in the program.

```java
//The public method that implemented the FIFO page-replacement algorithm.
public static void implementation(int[] reference_string, int frame_size) {

    //Displaying the whole reference string and user's input frame size.
    //General Notice: Dashed lines inserted in the print method is only for aesthetics purposes.
    System.out.println("-------------------------------------------------------------------------------------");
    //The Arrays.toString() method is used to display the reference string properly.
    System.out.println("Reference String: " + Arrays.toString(reference_string));
    System.out.println("Frame Size: "+ frame_size);
    System.out.println("-------------------------------------------------------------------------------------");
```

Figure 3 – Displaying Reference String and Frame Size

Implementation method is the "brain" of the program. The algorithm implementation occurs in this method. Additionally, this method has parameters. The parameters contain array and integer values from the program's main method. These parameters' significance will reflect to the output of reference string and frame size. Based on the initial requirements, printing the reference string and frame size is a must to determine which page numbers to be inserted in the Queue (Figure 4) and validate the frame size of the memory.

```java
    /* Declaration of a queue for the Algorithm. There are specific types of queue in Java
    but Linked List is utilized for this particular program. */
    Queue<Integer> integerQueue = new LinkedList<>();
```

Figure 4 – Initialization of Queue

Queue is an essential data structure in the algorithm since it implements "First-In and First-Out", similarly implementing the idea of the aforementioned page-replacement algorithm. The queue class have different implementations, but LinkedList is utilized to efficiently insert and remove elements in the queue. The developer decided to utilize *LinkedList* implementation to execute specialized methods such as offer() and poll().

```java
    // Declaration of initial values for Page Faults and Hits.
    int pageFaults= 0;
    int hits= 0;
```

Figure 5 – Declaration of initial values for Page Faults

Tracking the page faults and hits are requirements for testifying the algorithm's efficiency. These integer values will be utilized later within the program. Declaring these values to make sure that the program has a valid and invalid bit checker. Valid bit of the program will be referred to as "Hits." On the other hand, invalid bit will be referred to as "Page Fault."

```java
/*Declaration of the "for each loop" to simplify the execution of the integerQueue.
Take note: For each loop is specialized for iterating elements in an array or collection.*/
for(int page :reference_string){
     /*The if condition checks, if the current reference string index is visible within the queue.
    The contains() method of queue accomplish the job itself. After verifying the condition, ...*/
   if(integerQueue.contains(page)){
       //...incrementation of the hits variable will be initialized,...
       hits++;
       //...print the current queue and indicates the execution is a hit.
       System.out.print("Reference Page: " + page + " " + integerQueue);
       System.out.println(" -> (This execution is a hit.)");
   }
```

Figure 6 – For each loop and hit checker (Valid Bit Checker)

In this block of code is where the algorithm is specifically implemented. For each loop is utilized to check the index value of the array declared. The integer variable “page” will specify the current index value of the array “reference string.” If a specific page contains within the queue, the program will increment the value of hits (Valid Bit) and print the status of the queue indicating that there is no page fault.

```java
       /*The else condition is where the real process execute. In the program, an if condition
      is located inside it. It verifies if the queue size is equal to the frame_size value. After
      verification, it will... */
      else {
          if (integerQueue.size() == frame_size) {
              //...remove the first element within the queue,...
              integerQueue.poll();
          }
          //...add the current index value (page) of the reference string in queue's last position,...
          integerQueue.offer(page);
          //... increase the value of pageFaults variable and...
          pageFaults++;
          //...print the current queue status (indicating that the new page is added to the queue).
          System.out.println("Reference Page: " +  page + " " + integerQueue);
      }
  }
```

Figure 7 – FIFO process and Page Fault (Invalid Bit Checker)

The else condition implemented the enqueue and dequeue process within the LinkedList and simultaneously implementing the “FIFO” of the aforementioned algorithm. If the queue size is already equal to the frame size, it will remove the first element of the queue. Then, the queue will add the requested page by the offer() method. Additionally, increasing the value of the Page Fault (Invalid Bit).

```
62          System.out.println("-------------------------");
63              /* Displaying the number of Page Faults and Hits.
64          For your information, the algorithm is more efficient,
65          if page faults occurred less in the program. */
66          System.out.println("Number of Page Faults: "+pageFaults);
67          System.out.println("Number of Hits: "+hits);
```

Figure 8 – Total Number of Page Faults and Hits

Another way of testifying the algorithm's efficiency (purpose) is to display (functionality) the number of page faults (invalid bits) and hits (valid bits). The variables will return and displayed the incremented value of the variable. In addition, the dashed lines inserted in the print statement is just used for aesthetic purposes only.

```
69          //Declaration and initialization of total variable.
70          int total= pageFaults + hits;
71            /* Displaying the total value. Take note: The total value must be equal to
72           the size of the reference string.*/
73          System.out.println("Total: "+ total);
```

Figure 9 – Total Number of Reference Pages

Calculating and displaying the total number of pages in a program is to validate initial requirement given in the case study. According to the initial requirement, the reference string length should be 20 pages long. The program could get the number of total pages by adding the number of page faults and hits. In other words, this block of code is utilized for validation.

```
75          //Converting the value of pageFaults into double.
76          Double conPF = (double)pageFaults;
77          //Converting the value of total into double.
78          Double conTotal = (double)total;
79          //Converting the value of Hits into double.
80          Double conHits = (double) hits;
```

Figure 10 – Conversion of Integer Values to Double

Declaring variables with a data type of double will be essential for calculating the hit ratio and page fault ratio (Figure 11). Converting the declared integer variables into double will make the hit ratio and page fault ratio (Figure 11) value more accurate and precise. Converting the total variable into double is just a choice of the developer.

```
82          //Calculating the fault ratio and hit ratio of the algorithm.
83           double fault_ratio= conPF/conTotal;
84          double hit_ratio= conHits/conTotal;
```

Figure 11 – Hit Ratio and Page Fault Ratio (Additonal Features)

In page replacement algorithms, there are two metrics needed to be considered to analyze the algorithm's efficiency. The page fault rate also known as fault ratio refers to the ratio of page faults to the total number of pages. On the other hand, hit ratio refers to the ratio of page hits [7]. The developer has a resolution to include this feature to prove that the algorithm is effective in utilizing the available slot of memory. In other words, is the algorithm an effective memory manager?

```
86          //Displaying the value of Fault Ratio and Hit Ratio.
87          System.out.println("Fault ratio: " + df.format(fault_ratio));
88          System.out.println("Hit ratio: " + df.format(hit_ratio));
89          System.out.println("--------------------------");
90      }
91  }
```

Figure 12 - Displaying the value of Fault Ratio and Hit Ratio

Displaying the value of Fault Ratio and Hit Ratio is a must to provide a logical conclusion to the algorithm. If the hit ratio is higher than the fault ratio, then the algorithm is a sufficient memory manager. If the page fault ratio is higher, then the algorithm is not sufficient.

```
93  //Main class of the Java program.
94  public class FIFO_Page_Replacement_Algorithm {
95      //Main method of the Java Program.
96      public static void main (String[] args){
97          //This block is just the header of the program, indicating the program's name.
98          System.out.println("-----------------------------------------");
99          System.out.println("VIRTUAL MEMORY MACHINE MANAGER - FIFO edition");
100         System.out.println("-----------------------------------------");
```

Figure 13 – Main Class and Main Method

The main class and main method will be responsible for running the entire program. Inside the main method there are dashed lines inserted into two print statements. Another print statement is referring to the program's name. The purpose of the print statements is to serve as an aesthetic header for the program itself. Additionally, it will serve as the header of the program.

```
102         //Creating an object instantiation to access the method within the class "FIFO_algorithm".
103         FIFO_algorithm page = new FIFO_algorithm();
```

Figure 14 – Object Instatiation of FIFO_algorithm class

The object created in the main class provides a way to access the FIFO_algorithm class (Figure 2). This feature provides an opportunity to call the implementation() method (Figure 22) in the aforementioned class.

```
105         //Declaration of Scanner class for user prompt in frame size:
106         Scanner scan = new Scanner(System.in);
107
108         //Declaration of Random class to generate a random from 0 to 9
109         Random num = new Random();
```

Figure 15 – Declaration of Scanner and Random Classes

This block of code only pertains to the declaration of random and scanner classes. Random classes play an important role in generating the random page numbers in the reference. On the other hand, scanner classes play an important role in user input for the frame size.

```
111         //Declaration of Integer Values
112         int attempts = 0;
113         final int m_attempt = 2;
114         int frame = -1;
```

Figure 16 - Integer variables with values

Declaring the integer values for this block of code is essential for the user input. Attempt variable refers to the number of attempts accomplished by the user. The "m_attempt" variable refers to the maximum number of attempts, the declared value for it is 2. Lastly, int frame refers to the size of the memory frame.

```
116         //If the attempt is less than 2, it will execute the following:
117         while (attempts < m_attempt){
118             //Indication that the user must enter their preferred frame size for the queue.
119             System.out.print("Enter your preferred frame size (3 to 5 only): ");
120             frame = scan.nextInt();
```

Figure 17 – While Condition and User Input

The while condition in this block of code executes when the number of attempts is less than the value of max attempts. In addition, this code section allows the user to input the frame size number three, four, or five.

```
122             if (frame<=5 && frame>=3) {
123                 break; // If the frame size number is valid, the program will exit the loop immediately.
124             }
125             else {
126                 attempts ++; // If not, the attempt value will be incremented and...
127                 if(attempts < m_attempt) {
128                     // will ask the user to input a number again.
129                     System.out.println("Invalid input, try again.");
130                 }
131             }
```

Figure 18 – Frame Size Checker

If the user input of the frame size is greater than 3 and less than 5, then the program will exit the loop immediately and proceed to the algorithm execution. If the user input is not valid, the attempts value will be incremented and at the same time will ask the user to input a number for the frame size again.

```
135         // If the frame size does not meet the qualification, it will display the following: "
136          if (frame < 3 || frame > 5){
137             System.out.println("-------------------------");
138             System.out.println("FRAME SIZE - OUT OF BOUNDS");
139             System.out.println("-------------------------");
140 
141             //Indicating that the program is ended.
142             System.out.println("------------------");
143             System.out.println("PROGRAM TERMINATED");
144             System.out.println("------------------");
145             return;
146         }
```

Figure 19 – Out of Bounds Indicator

If the user tries to input an invalid number again. The program will indicate that the frame size is out of bounds and at the same time, the program will exit immediately after indicating that the program is terminated.

```
149         int ref_size = 20;
150
151         //Declaration of the array of Reference String and indicates the size of the array is ref_size.
152         int[] ref_page = new int[ref_size];
```

Figure 20 - Reference Size and Page Declaration

After exiting the loop immediately (Figure 19), the program will declare first the array size with a variable named ref_size and immediately creates the array ref_page. This array is utilized for creating the reference string of the program.

```
154     //For loop is initialized in this block to generate an array automatically with random numbers.
155     for (int i = 0; i < ref_size; i++) {
156         /*Random generation numbers (from 0 to 9) will be
157         processed within this particular line of code. */
158         ref_page[i] = num.nextInt( bound: 9);
159     }
```

Figure 21 - Random Numbers Generation

A for loop is initialized and utilized to generate different page numbers in the reference string. Random numbers will be generated inside of the for loop. Bound indicates that the random class (Figure 15) will only generate numbers from 0 to 9. With this feature in Java™, a requirement has been accomplished and executed in the program.

```
160         /*Accessing the method with parameters within the class "FIFO_algorithm"
161         by using object instantiation. */
162         page.implementation(ref_page, frame);
163         //Indicating that the program is ended.
164         System.out.println("------------------");
165         System.out.println("PROGRAM TERMINATED");
166         System.out.println("------------------");
167     }
168 }
```

Figure 22 - Method Calling and Program Termination

The implementation method has parameters for reference string and frame size. The declared values from this class (Figure 13) will be fetched into the FIFO_algorithm class (Figure 3) by calling the implementation method. After the method is executed, the program will terminate.

Results

Evaluating the results of the simulation (program) will determine if the aforementioned algorithm is sufficient enough to manage memory with different frame sizes. These results will ensure and testify the aforementioned algorithm's effectiveness and reliability. This chapter uncovers the page fault occurrences within the simulation.

```
-----------------------------------------
VIRTUAL MEMORY MACHINE MANAGER - FIFO edition
-----------------------------------------
Enter your preferred frame size (3 to 5 only): 3
--------------------------------------------------------------------------------
Reference String: [7, 4, 4, 3, 4, 2, 4, 3, 8, 4, 3, 0, 6, 4, 4, 5, 0, 6, 1, 0]
Frame Size: 3
--------------------------------------------------------------------------------
Reference Page: 7 [7]
Reference Page: 4 [7, 4]
Reference Page: 4 [7, 4] -> (This execution is a hit.)
Reference Page: 3 [7, 4, 3]
Reference Page: 4 [7, 4, 3] -> (This execution is a hit.)
Reference Page: 2 [4, 3, 2]
Reference Page: 4 [4, 3, 2] -> (This execution is a hit.)
Reference Page: 3 [4, 3, 2] -> (This execution is a hit.)
Reference Page: 8 [3, 2, 8]
Reference Page: 4 [2, 8, 4]
Reference Page: 3 [8, 4, 3]
Reference Page: 0 [4, 3, 0]
Reference Page: 6 [3, 0, 6]
Reference Page: 4 [0, 6, 4]
Reference Page: 4 [0, 6, 4] -> (This execution is a hit.)
Reference Page: 5 [6, 4, 5]
Reference Page: 0 [4, 5, 0]
Reference Page: 6 [5, 0, 6]
Reference Page: 1 [0, 6, 1]
Reference Page: 0 [0, 6, 1] -> (This execution is a hit.)
```

Figure 23 - Output 1 (Frame Size 3)

Output 1 is evaluating what would be the result if the simulated main memory has a frame size of three. Based on the result, the developer analyzed and evaluated that the presence of the page faults is dominant in this scenario. Page fault occurred when the requested page is not available in the simulated main memory. In addition, the output of

the program is quite different in the theoretical approach within the algorithm. To add context, the arrangement of numbers in the program is different from the theoretical application. In theoretical application, the number arrangement of fourth reference page (Figure 23) will be 2, 4, and 3. However in the program, the arrangement of numbers is 4, 3, and 2. The reason is that the Queue class in Java™ cannot modify the element's value directly, and the developer based the initial requirements' output in a website called *Prepinsta* [8].

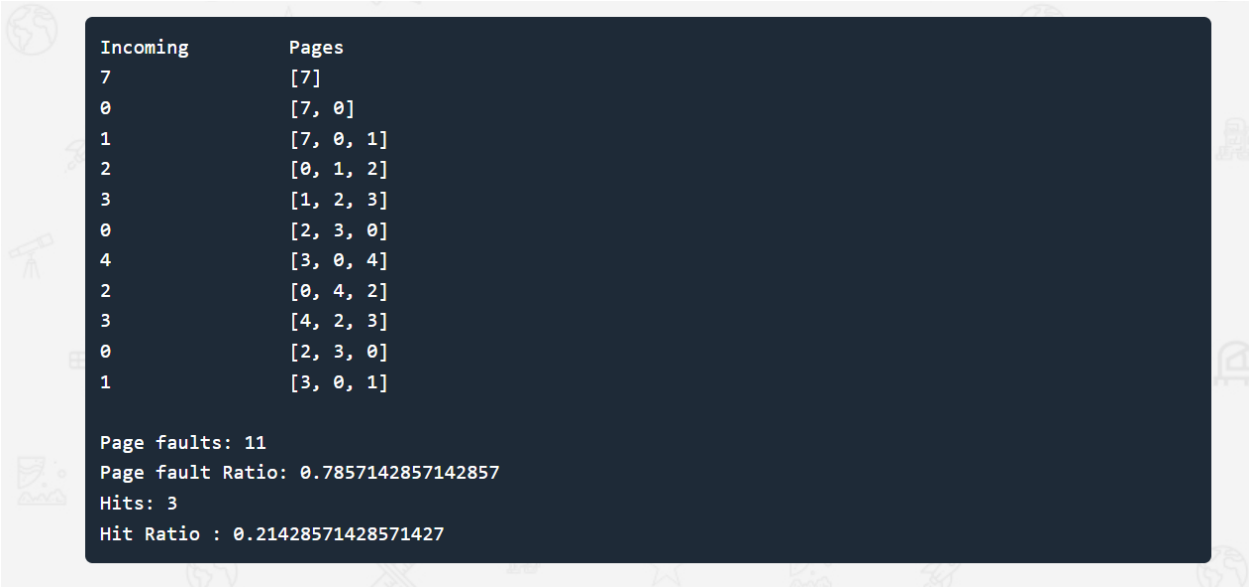

Figure 24 – FIFO Algorithm in Java (Prepinsta) [8]

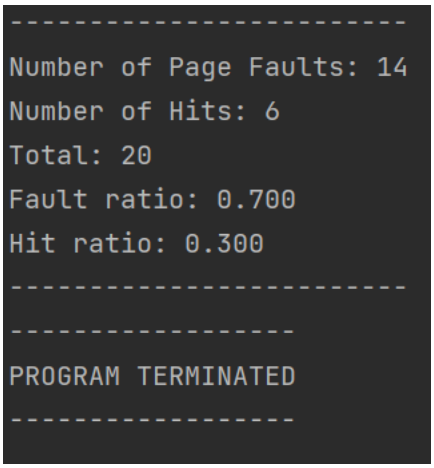

Figure 25 - Output 1 Results (Frame Size 3)

Output 1 results further evaluate and testify the effectiveness and reliability of the aforementioned algorithm. Number of page faults already being dominant in the first part of the output 1 (Figure 24). The staggering difference between fault ratio and hit ratio

arrived at the conclusion that the aforementioned algorithm is NOT effective and reliable in managing memory with small frame sizes.

```
-----------------------------------------
VIRTUAL MEMORY MACHINE MANAGER - FIFO edition
-----------------------------------------
Enter your preferred frame size (3 to 5 only): 4
--------------------------------------------------------------------------------
Reference String: [4, 0, 6, 3, 1, 6, 2, 6, 8, 8, 4, 1, 1, 1, 7, 1, 3, 3, 5, 6]
Frame Size: 4
--------------------------------------------------------------------------------
Reference Page: 4 [4]
Reference Page: 0 [4, 0]
Reference Page: 6 [4, 0, 6]
Reference Page: 3 [4, 0, 6, 3]
Reference Page: 1 [0, 6, 3, 1]
Reference Page: 6 [0, 6, 3, 1] -> (This execution is a hit.)
Reference Page: 2 [6, 3, 1, 2]
Reference Page: 6 [6, 3, 1, 2] -> (This execution is a hit.)
Reference Page: 8 [3, 1, 2, 8]
Reference Page: 8 [3, 1, 2, 8] -> (This execution is a hit.)
Reference Page: 4 [1, 2, 8, 4]
Reference Page: 1 [1, 2, 8, 4] -> (This execution is a hit.)
Reference Page: 1 [1, 2, 8, 4] -> (This execution is a hit.)
Reference Page: 1 [1, 2, 8, 4] -> (This execution is a hit.)
Reference Page: 7 [2, 8, 4, 7]
Reference Page: 1 [8, 4, 7, 1]
Reference Page: 3 [4, 7, 1, 3]
Reference Page: 3 [4, 7, 1, 3] -> (This execution is a hit.)
Reference Page: 5 [7, 1, 3, 5]
Reference Page: 6 [1, 3, 5, 6]
```

Figure 26 - Output 2 (Frame Size 4)

Output 2 is evaluating what would be the result if the simulated main memory has a frame size of four. Based on the result, the developer analyzed and evaluated that the presence of the page faults is STILL dominant in this scenario. In addition, the output shows the number of hits (Valid Bits) is incremented in this scenario.

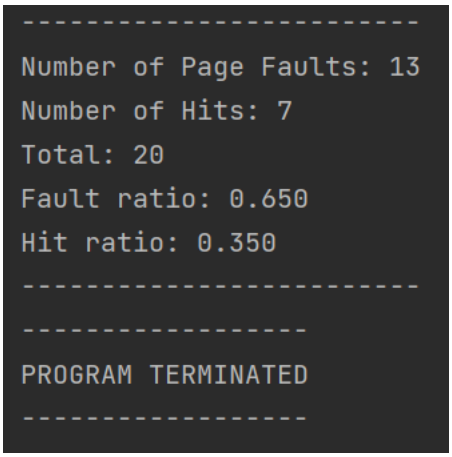

Figure 27 - Output 2 Results (Frame Size 4)

Output 2 results further evaluate and testify the effectiveness and reliability of the aforementioned algorithm. Number of page faults already still being dominant in the first part of the output 2 (Figure 27). The minute difference between fault ratio and hit ratio arrived at the conclusion that the aforementioned algorithm is STILL not effective and reliable in managing memory with average frame sizes. Logically speaking, the fault (Invalid Bit) ratio is still greater than the hit (Valid Bit) ratio.

```
-----------------------------------------
VIRTUAL MEMORY MACHINE MANAGER - FIFO edition
-----------------------------------------
Enter your preferred frame size (3 to 5 only): 5
--------------------------------------------------------------------------------
Reference String: [0, 0, 8, 0, 4, 1, 6, 0, 7, 6, 0, 3, 0, 3, 4, 0, 4, 1, 0, 0]
Frame Size: 5
--------------------------------------------------------------------------------
Reference Page: 0 [0]
Reference Page: 0 [0] -> (This execution is a hit.)
Reference Page: 8 [0, 8]
Reference Page: 0 [0, 8] -> (This execution is a hit.)
Reference Page: 4 [0, 8, 4]
Reference Page: 1 [0, 8, 4, 1]
Reference Page: 6 [0, 8, 4, 1, 6]
Reference Page: 0 [0, 8, 4, 1, 6] -> (This execution is a hit.)
Reference Page: 7 [8, 4, 1, 6, 7]
Reference Page: 6 [8, 4, 1, 6, 7] -> (This execution is a hit.)
Reference Page: 0 [4, 1, 6, 7, 0]
Reference Page: 3 [1, 6, 7, 0, 3]
Reference Page: 0 [1, 6, 7, 0, 3] -> (This execution is a hit.)
Reference Page: 3 [1, 6, 7, 0, 3] -> (This execution is a hit.)
Reference Page: 4 [6, 7, 0, 3, 4]
Reference Page: 0 [6, 7, 0, 3, 4] -> (This execution is a hit.)
Reference Page: 4 [6, 7, 0, 3, 4] -> (This execution is a hit.)
Reference Page: 1 [7, 0, 3, 4, 1]
Reference Page: 0 [7, 0, 3, 4, 1] -> (This execution is a hit.)
Reference Page: 0 [7, 0, 3, 4, 1] -> (This execution is a hit.)
```

Figure 28 - Output 3 (Frame Size 5)

Output 3 is evaluating what would be the result if the simulated main memory has a frame size of five. Based on the result, the developer analyzed and evaluated that the presence of the page faults and hits are already EQUAL (in this scenario only). Arriving to the conclusion that the aforementioned algorithm could be semi-reliable if the frame-size of the memory is large.

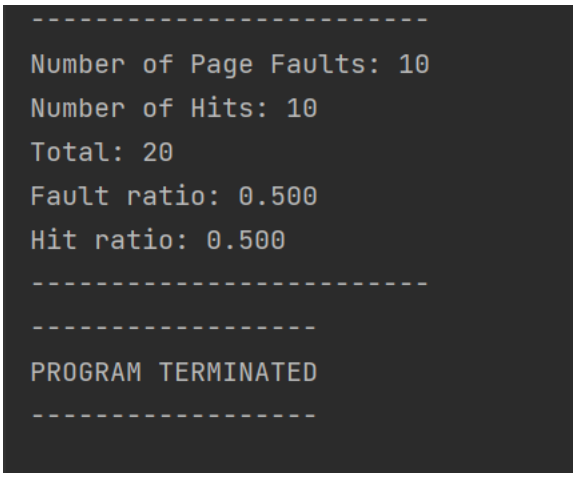

Figure 29 - Output 3 Results (Frame Size 5)

Output 3 results further evaluate and testify the effectiveness and reliability of the aforementioned algorithm. Number of page faults is already being equal to the number of hits as shown at the output picture (Figure 28). The similarity between fault ratio and hit ratio arrived at the conclusion that the aforementioned algorithm could be semi-effective and reliable in managing memory with average frame sizes. The evaluation for output 3 is depending to the output of reference string. If different values are inserted in the reference string, the number of page faults can be still dominant. In addition, the number of hits (Valid bits) is dependent to the frame size of the memory. However, the aforementioned algorithm FAILED to reduce the number of page faults.

A table is also created and utilized to evaluate the reliability and efficiency of the aforementioned algorithm. The developer formulated expected outputs. However, the actual output for output 3 are quite contradicting it. This table is another proof that the aforementioned algorithm is not efficient and reliable in memory management.

Table 1 - Simulation's Scenario Testing

Scenario Cases	Number of Pages	Frame Size Input	Expected Output	Actual Output
Output 1	20	3	Dominant presence of page faults.	Number of page faults is greater than valid bits. Passed
Output 2	20	4	Dominant presence of page faults.	Number of page faults is greater than valid bits.

				Passed
Output 3	20	5	Lesser number of page faults.	First Scenario: Number of page faults and hits are equal. Failed Second Scenario: Number of page faults is still dominant. Failed Third Scenario: Number of valid bits is higher, but the number of page faults is the predominant factor in the current simulation. Failed

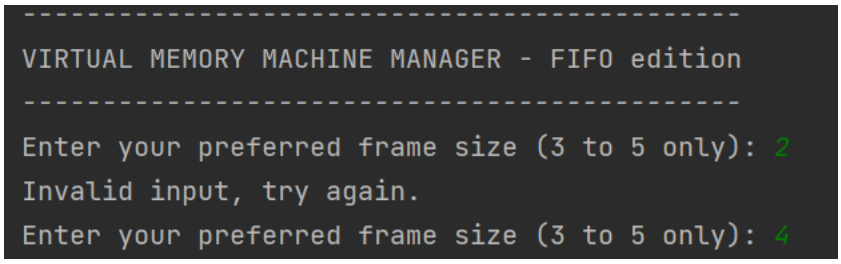

Figure 30 - Output 4 (Incorrect First Input, Correct Second Input)

Output 4 evaluates if the additional feature in the program is functioning as intended. The user is able to input another number if their previous input does not conform to the qualifications of the program. This figure just shows that the additional feature is working as intended.

```
Reference Page: 2 [2]
Reference Page: 2 [2] -> (This execution is a hit.)
Reference Page: 3 [2, 3]
Reference Page: 0 [2, 3, 0]
Reference Page: 2 [2, 3, 0] -> (This execution is a hit.)
Reference Page: 6 [2, 3, 0, 6]
Reference Page: 6 [2, 3, 0, 6] -> (This execution is a hit.)
Reference Page: 7 [3, 0, 6, 7]
Reference Page: 8 [0, 6, 7, 8]
Reference Page: 5 [6, 7, 8, 5]
Reference Page: 6 [6, 7, 8, 5] -> (This execution is a hit.)
Reference Page: 8 [6, 7, 8, 5] -> (This execution is a hit.)
Reference Page: 2 [7, 8, 5, 2]
Reference Page: 1 [8, 5, 2, 1]
Reference Page: 8 [8, 5, 2, 1] -> (This execution is a hit.)
Reference Page: 8 [8, 5, 2, 1] -> (This execution is a hit.)
Reference Page: 2 [8, 5, 2, 1] -> (This execution is a hit.)
Reference Page: 6 [5, 2, 1, 6]
Reference Page: 3 [2, 1, 6, 3]
Reference Page: 1 [2, 1, 6, 3] -> (This execution is a hit.)
------------------------
Number of Page Faults: 11
Number of Hits: 9
Total: 20
Fault ratio: 0.550
Hit ratio: 0.450
------------------------
```

Figure 31 - Output 4 Results

This figure only validates that the program will function properly if the second user input is correct. After the execution of the aforementioned algorithm, the program will terminate.

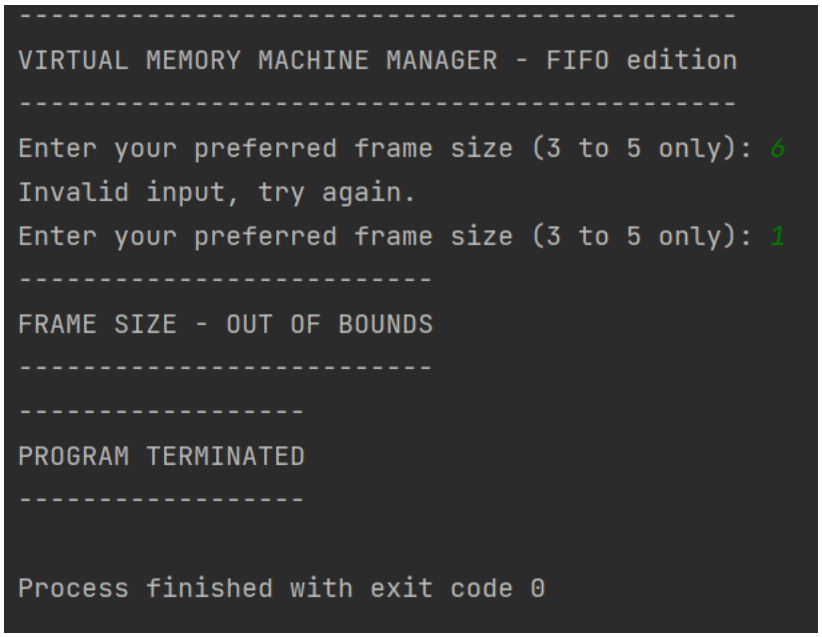

Figure 32 - Output 5 (Incorrect first and second inputs)

Output 5 shows a use-case scenario when the user is unable to input numbers that conforms to the program qualification. The program will indicate that the frame size for the simulated main memory is out of bounds. In other words, if the number is greater than 5 and less than 3. Then after the indication, the program will terminate.

Challenges and Solutions

Skill Decay – the first challenge that has been encountered during the creation of the case study. Skill decay (also known as skill degradation) refers to the loss of proficiency in a skill due to insufficient practice [9]. In other words, a programmer can once acquire and learn about a programming language but, if a programmer never does coding practices and does not utilize the programming language for a long time, he/she will have difficulty to retrieve or recall knowledge about the domain.

The best solution for this particular issue is to practice and have a refresher by browsing information on the Internet. To emphasize, the case study itself became an opportunity to recall and refresh the skills to programming language (Java™) since the developer applied the aforementioned algorithm. In addition, internet browsing became helpful because the developer is able to find new coding information that can be applied to the simulation (program).

Incorrect Results – the second challenge that has been encountered during the creation of the case study. Incorrect results are a big *"NO"* to this case study because the developer will evaluate the reliability and effectiveness of the aforementioned page-replacement algorithm. The primary cause of incorrect result is the flawed logic of the program. In addition, incorrect results consequently lead to incorrect evaluations and that will lead to incorrect conclusions. The best solution for this particular issue is to trace the flawed logic of the program and accomplish the process of debugging.

Code Inefficiency and Redundancy - the third challenge that has been encountered during the creation of the case study. During the creation of the simulation or program, the developer encountered that the first three steps of the algorithm execution is correct. However, the program does not execute enqueue and dequeue (add() and remove()) operations when a page fault occurred.

After encountering the problem, the developer has a solution. The developer introduced a block of redundant codes. The redundant code successfully executed the algorithm, but the reference page number string has defined numbers, that feature in the program contradicted the initial requirement of the case study (random generation).

If different numbers are inserted in the reference string, this situation would break down the algorithms' logic. The best solution for this issue was to initialize for each loop (Figure 6 – Code Explanation) since the reference string is an array. This approach ensures random generation compatibility and removes code duplication. To preserve the algorithm's logic, the developer initialized an if-else statement (Figure 6 and 7 – Code Explanation) to verify if the reference page execution has a Valid Bit (Hits) (Figure 6 – Code Explanation) or Invalid Bit (Page Fault) (Figure 7 – Code Explanation). The program reduced redundancy and became more efficient with the help of these approaches.

Conclusion

The overarching outcome of the case study provides the developer to learn and gain understanding about the complexity of the aforementioned algorithm. Key findings such as the impact of page frame sizes affecting the number of page fault became a revelation of this study. In addition, the evaluation of outputs significantly provides logical conclusions to the effectiveness and reliability of the algorithm itself.

In evaluating the strengths and weaknesses of the algorithm, the developer concluded that the algorithm's strength is the easy implementation. The aforementioned algorithm's logic is simple because the algorithm itself will replace the oldest page with a new one if a page fault occurs. In other words, faster implementation of the aforementioned algorithm to the programming language. However, the aforementioned algorithm has a drawback of being vulnerable to the Belady's Oddity [10]. Belady's anomaly refers to the phenomenon in operating systems in where increasing the number of page frames in memory increases the number of page faults [11]. This phenomenon happens in the aforementioned algorithm. The output presented below is a sufficient proof that the algorithm suffers from this anomaly in certain scenarios.

```
Reference String: [5, 1, 7, 3, 0, 8, 5, 8, 8, 2, 3, 6, 7, 2, 6, 8, 4, 4, 6, 6]
Frame Size: 5
--------------------------------------------------------------------------------
Reference Page: 5 [5]
Reference Page: 1 [5, 1]
Reference Page: 7 [5, 1, 7]
Reference Page: 3 [5, 1, 7, 3]
Reference Page: 0 [5, 1, 7, 3, 0]
Reference Page: 8 [1, 7, 3, 0, 8]
Reference Page: 5 [7, 3, 0, 8, 5]
Reference Page: 8 [7, 3, 0, 8, 5] -> (This execution is a hit.)
Reference Page: 8 [7, 3, 0, 8, 5] -> (This execution is a hit.)
Reference Page: 2 [3, 0, 8, 5, 2]
Reference Page: 3 [3, 0, 8, 5, 2] -> (This execution is a hit.)
Reference Page: 6 [0, 8, 5, 2, 6]
Reference Page: 7 [8, 5, 2, 6, 7]
Reference Page: 2 [8, 5, 2, 6, 7] -> (This execution is a hit.)
Reference Page: 6 [8, 5, 2, 6, 7] -> (This execution is a hit.)
Reference Page: 8 [8, 5, 2, 6, 7] -> (This execution is a hit.)
Reference Page: 4 [5, 2, 6, 7, 4]
Reference Page: 4 [5, 2, 6, 7, 4] -> (This execution is a hit.)
Reference Page: 6 [5, 2, 6, 7, 4] -> (This execution is a hit.)
Reference Page: 6 [5, 2, 6, 7, 4] -> (This execution is a hit.)
-------------------------
Number of Page Faults: 11
Number of Hits: 9
Total: 20
Fault ratio: 0.550
Hit ratio: 0.450
```

Figure 33 - Proof of Belady's Anomaly

Evaluating the program's output in the previous figure (Figure 33), the aforementioned page-replacement algorithm fails to reduce the number of page faults in certain scenarios. The frame size generally impacts the number of page faults by reducing it. However, in certain scenarios, the aforementioned page-replacement algorithm fails to accomplish its purpose making the algorithm inefficient and unreliable. If the developer is given more time and resources, the best suggestion would be to apply the aforementioned algorithm in different programming languages (such as Python, C#, C++, Perl etc.) simultaneously. In addition, it would be a great idea to compare an implemented page-replacement algorithm to another algorithm. This analysis would make the developer or researcher to gain more insights about the strengths and weaknesses of the algorithm.

References:

[1] A. S. Chavan, K. R. Nayak, K. D. Vora, M. D. Purohit, and P. M. Chawan, "A comparison of page replacement algorithms," International Journal of Engineering and Technology, vol. 3, no. 2, pp. 171–174, 2011. doi:10.7763/ijet.2011.v3.218

[2] G. Rexha, E. Elmazi, and I. Tafa, "A comparison of three-page replacement algorithms: FIFO, LRU and optimal," *Academic Journal of Interdisciplinary Studies*, Aug. 2015. doi:10.5901/ajis.2015.v4n2s2p56

[3] J. Gosling, The Java language specification: Java SE 7. Upper Saddle River, Nj: Addison-Wesley, 2013.

[4] W3Schools, "Java Arrays," W3schools.com, 2019. https://www.w3schools.com/java/java_arrays.asp

[5] "Queue Interface in Java - GeeksforGeeks," GeeksforGeeks, Jun. 30, 2016. https://www.geeksforgeeks.org/queue-interface-java/

[6] "Java.util.Random class in Java," GeeksforGeeks, May 26, 2017. https://www.geeksforgeeks.org/java-util-random-class-java/

[7] "Learn about common methods and metrics for evaluating page replacement algorithms in operating systems, such as page fault rate, Belady's anomaly, hit ratio, and average access time.," Linkedin.com, Mar. 10, 2024. https://www.linkedin.com/advice/0/how-can-you-accurately-measure-page-replacement-eezcf

[8] "FIFO Page Replacement Algorithm in Java | Prepinsta," PREP INSTA, Jan. 21, 2023. https://prepinsta.com/operating-systems/page-replacement-algorithms/fifo/fifo-in-java/ (accessed May 01, 2025).

[9] M. Klostermann, S. Conein, T. Felkl, and A. Kluge, "Factors Influencing Attenuating Skill Decay in High-Risk Industries: A Scoping Review," Safety, vol. 8, no. 2, p. 22, Mar. 2022, doi: https://doi.org/10.3390/safety8020022.

[10] S. H. Abbas, W. A. K. Naser, and L. M. Kadhim, "Study and Comparison of Replacement Algorithms," International Journal of Engineering Research and Advanced Technology, vol. 08, no. 08, pp. 01–06, 2022, doi: https://doi.org/10.31695/ijerat.2022.8.8.1.

[11] "Belady's Anomaly in Page Replacement Algorithms," GeeksforGeeks, Jul. 13, 2018. https://www.geeksforgeeks.org/beladys-anomaly-in-page-replacement-algorithms/