# Page Replacement Algorithm Report

Name: Anass Agarrab Student Id: L202126100103

## Contents



# 1 Introduction

**General Introduction to Page Replacement Algorithms:**

Page replacement algorithms are a cornerstone of memory management in computer operating systems. Their primary role is to decide which memory pages to swap out or replace when a page fault occurs, typically when a program requests a page that is not in the main memory. These algorithms aim to minimize the number of page faults, thereby improving the efficiency of memory usage. Common strategies include FIFO (First-In, First-Out), LRU (Least Recently Used), Optimal, and Clock algorithms. Each of these algorithms has a unique approach to managing memory, balancing factors like recency of access, frequency of use, and predictive analysis of future requests. Implementing effective page replacement strategies is crucial for optimizing system performance, especially in environments with limited memory resources.

**General Introduction to the Page Replacement Program:**

This program is designed to simulate various page replacement algorithms in a computing environment. Developed in C++, it provides a practical tool for understanding how different algorithms manage memory under varying conditions. The program was developed using DEV-C++, a popular integrated development environment that simplifies C++ development. To compile and run this program, the MinGW compiler is used, which provides a suite of GNU tools, including g++ for compiling C++ code. The program allows users to choose between different algorithms like FIFO, LRU, Optimal, and Clock, and input a series of page references and the number of frames available in memory. It then simulates the chosen page replacement strategy, providing insights into each algorithm's behavior and efficiency. This tool is particularly useful for students and professionals interested in operating systems, memory management, and algorithmic efficiency.

# 2 Design

### 2.1. Input/Output Module:

- This module is responsible for reading user inputs (like the choice of algorithm, number of frames, and page reference string) and displaying the outputs (like page faults and frame status after each page reference).

### 2.2. Workflow:

- The program has distinct workflows for different algorithms (FIFO, LRU, Optimal, CLOCK). Each algorithm follows a specific logic to decide which page to replace when a page fault occurs.

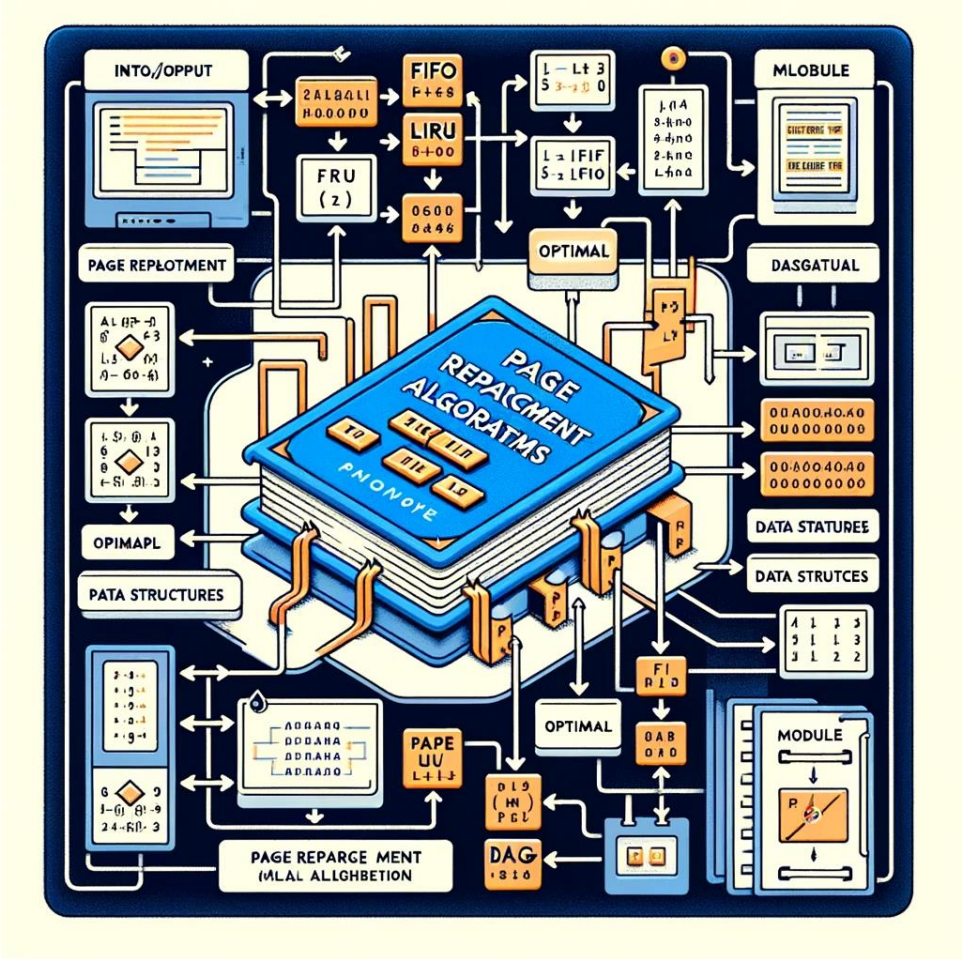
### 2.3. Data Structures:

- Various data structures are utilized:
- Arrays or Lists: To store the page reference string and the current status of frames.
- Queue (in FIFO): To track the order of pages.
- Linked List (in LRU): To maintain the recently used pages.
- Set or Boolean Array (in CLOCK): To track whether a page was recently accessed.

### 2.4. Modules:

- User Input Module: Handles the parsing and validation of user inputs.
- Data Structure Preparation Module: Sets up necessary data structures based on the chosen algorithm.
- Page Replacement Logic Module: Contains the core logic for each page replacement algorithm.

The attached figure illustrates the overall architecture and workflow of the program, highlighting how these components interact with each other:




This design ensures modularity and separation of concerns, making the program easier to understand, maintain, and extend.

# 3 Implementation

The Page Replacement Algorithms program leverages various key data structures and algorithms to simulate different strategies for managing pages in memory. Here's a detailed explanation of each:

**3.1. FIFO (First-In, First-Out):**

- **Data Structure:** Queue
- **Algorithm Explanation:** FIFO maintains the order in which pages enter memory. When a page fault occurs, the algorithm removes the oldest page (the one at the front of the queue) and inserts the new page at the back. This approach is simple but doesn't account for the frequency or recency of page accesses.

```cpp
void pageFaultFIFO(const std::vector<char>& pages, int capacity) {
    std::queue<char> frameQueue;
    std::vector<char> frames(capacity, '-'); // Initialize frames with '-' indicating empty
    int pageFaults = 0;
    for (char page : pages) {
        std::cout << "Sequence Number: " << &page - &pages[0] << "\n";
        std::cout << "Reference: " << page << "\n";

        // Check if the page is already in the frame
        auto it = std::find(frames.begin(), frames.end(), page);
        bool isPageFault = (it == frames.end());
        // If it is a page fault and there is no room, remove the oldest page
        if (isPageFault && frameQueue.size() == capacity) {
            char oldPage = frameQueue.front();
            frameQueue.pop();
            *std::find(frames.begin(), frames.end(), oldPage) = page; // Replace the old page with the new page
        } else if (isPageFault) {
            // If it is a page fault and there is room, add the new page
            frames[frameQueue.size()] = page;
        }
        // Add the new page to the queue if it's a page fault
        if (isPageFault) {
            frameQueue.push(page);
            pageFaults++;
            std::cout << "Page Fault: true\n";
        } else {
            std::cout << "Page Fault: false\n";
        }
        // Display the frame status
        for (int i = 0; i < capacity; ++i) {
            std::cout << "Frame " << i + 1 << ": ";
            if (frames[i] != '-') {
                std::cout << frames[i];
            } else {
                std::cout << "null";
            }
            if (isPageFault && frameQueue.front() == frames[i]) {
                std::cout << " (victim)";
            }
            std::cout << "\n";
        }
        std::cout << "\n"; // Newline for formatting
    }
    std::cout << "Total page faults: " << pageFaults << "\n";
}
```

### 3.2. LRU (Least Recently Used):

- **Data Structure:** Linked List, HashMap

- **Algorithm Explanation:** LRU tracks page usage and prioritizes pages based on how recently they were accessed. When a page is accessed or brought into memory, it is moved to the front of the list. If a page fault occurs and a replacement is needed, the least recently used page (the one at the end of the list) is removed. The HashMap is used for quick lookups to check if a page is in memory and to find its position in the list.

```cpp
void pageFaultLRU(const std::vector<char>& pages, int capacity) {
    std::list<char> framesList; // Stores the pages in the order of usage
    std::unordered_map<char, std::list<char>::iterator> pagesMap; // Maps each page to its position in framesList
    int pageFaults = 0;

    for (size_t i = 0; i < pages.size(); ++i) {
        std::cout << "Sequence Number: " << i << "\n";
        std::cout << "Reference: " << pages[i] << "\n";

        auto it = pagesMap.find(pages[i]);

        // Page not in frames, we have a page fault
        if (it == pagesMap.end()) {
            // Check if we need to remove the least recently used page
            if (framesList.size() == capacity) {
                char last = framesList.back();
                framesList.pop_back(); // Remove the least recently used page
                pagesMap.erase(last); // Remove it from the map
            }
            pageFaults++;
            std::cout << "Page Fault: true\n";
        } else {
            // Page is in frames, move it to the front to mark it as recently used
            framesList.erase(it->second);
            std::cout << "Page Fault: false\n";
        }

        // Add new page to the front of the list and in the map
        framesList.push_front(pages[i]);
        pagesMap[pages[i]] = framesList.begin();

        // Display the frame status
        int j = 0;
        for (auto page : framesList) {
            std::cout << "Frame " << j + 1 << ": " << page << "\n";
            j++;
        }
        for (; j < capacity; j++) {
            std::cout << "Frame " << j + 1 << ": null\n";
        }
        std::cout << "\n"; // Newline for formatting
    }

    std::cout << "Total page faults: " << pageFaults << "\n";
}
```

### 3.3. Optimal:

- **Data Structure:** Set, Array

- **Algorithm Explanation:** The Optimal algorithm requires future knowledge of page requests. It replaces the page that will not be used for the longest time in the future. This is determined by looking ahead in the page reference string and identifying the page to be accessed farthest in the future. Although theoretically the best in terms of minimizing page faults, it's impractical in real-world scenarios due to its requirement of future knowledge.

```cpp
int findOptimalVictim(const std::vector<char>& pages, std::vector<char>& frames, int currentPageIndex) {
    int index = -1;
    int farthest = currentPageIndex;
    for (int i = 0; i < frames.size(); ++i) {
        int j;
        for (j = currentPageIndex; j < pages.size(); ++j) {
            if (frames[i] == pages[j]) {
                if (j > farthest) {
                    farthest = j;
                    index = i;
                }
                break;
            }
        }
        if (j == pages.size()) {
            return i; // If the page isn't going to be used again, return this frame
        }
    }
    return (index == -1) ? 0 : index; // If all are going to be used, replace the last used
}
```

```cpp
void pageFaultOptimal(const std::vector<char>& pages, int capacity) {
    std::vector<char> frames(capacity, '-'); // Initialize with '-' indicating empty
    std::unordered_map<char, bool> isInFrame;
    int pageFaults = 0;

    for (size_t i = 0; i < pages.size(); ++i) {
        std::cout << "Sequence Number: " << i << "\n";
        std::cout << "Reference: " << pages[i] << "\n";

        if (!isInFrame[pages[i]]) {
            int victimIndex = -1;
            if (isInFrame.size() >= capacity) {
                victimIndex = findOptimalVictim(pages, frames, i + 1);
                isInFrame.erase(fram
            }

            for (int j = 0; j < capacity; ++j) {
                if (frames[j] == '-' || j == victimIndex) {
                    frames[j] = pages[i];
                    isInFrame[pages[i]] = true;
                    pageFaults++;
                    break;
                }
            }
            std::cout << "Page Fault: true\n";
        } else {
            std::cout << "Page Fault: false\n";
        }

        // Display the frame status
        for (int j = 0; j < capacity; ++j) {
            std::cout << "Frame " << j + 1 << ": ";
            if (frames[j] != '-') {
                std::cout << frames[j];
            } else {
                std::cout << "null";
            }
            std::cout << "\n";
        }
        std::cout << "\n"; // Newline for formatting
    }

    std::cout << "Total page faults: " << pageFaults << "\n";
}
```

int findOptimalVictim (const std::vector<char>& pages, std::vector<char>& frames, int currentPageIndex)

### 3.4. Clock (Second Chance):

- **Data Structure:** Circular List (ArrayList of custom PageFrame objects)

- **Algorithm Explanation:** The Clock algorithm is a practical implementation of LRU. It keeps track of pages using a circular list, with each page having an additional 'referenced' bit. The algorithm gives a second chance to pages before replacement. When a page fault occurs, it scans from the current position of a 'hand' through the circular list. If it encounters a page with the 'referenced' bit set to true, it gives it a second chance by setting this bit to false and moves to the next page. It replaces the first page it finds with the 'referenced' bit set to false.

```cpp
void pageFaultClock(const std::vector<char>& pages, int capacity) {
    std::vector<PageFrame> frameVector(capacity, {'-', false});
    int pageFaults = 0;
    int hand = 0;  // The "hand" of the clock
    for (size_t i = 0; i < pages.size(); ++i) {
        std::cout << "Sequence Number: " << i << "\n";
        std::cout << "Reference: " << pages[i] << "\n";
        // Check if the page is in any frame
        bool pageFound = false;
        for (auto& frame : frameVector) {
            if (frame.page == pages[i]) {
                frame.referenced = true; // Mark as referenced
                pageFound = true;
                break;
            }
        }
        if (!pageFound) {
            while (frameVector[hand].referenced) {
                frameVector[hand].referenced = false; // Unset the reference bit
                hand = (hand + 1) % capacity; // Move the hand forward
            }
            // Replace the page at the hand
            frameVector[hand].page = pages[i];
            frameVector[hand].referenced = true; // Set the reference bit
            hand = (hand + 1) % capacity; // Move the hand forward

            pageFaults++;
            std::cout << "Page Fault: true\n";
        } else {
            std::cout << "Page Fault: false\n";
        }
        // Display the frame status
        for (int j = 0; j < capacity; ++j) {
            std::cout << "Frame " << j + 1 << ": ";
            if (frameVector[j].page != '-') {
                std::cout << frameVector[j].page << (frameVector[j].referenced ? "*" : "");
            } else {
                std::cout << "null";
            }
            std::cout << "\n";
        }
        std::cout << "\n"; // Newline for formatting
    }

    std::cout << "Total page faults: " << pageFaults << "\n";
}
```

Each of these algorithms is implemented as a separate module in the program, ensuring modularity and ease of expansion or modification. The choice of data structures for each algorithm is crucial to its efficiency and effectiveness in simulating page replacement strategies.

# 4 Case study

Below are some sample screenshots of different ways to interact with your programs.

- Sample 1:

```
PS D:\anass> ./PageReplacement 3 5 a s d f r e
Sequence Number: 0
Reference: a
Page Fault: true
Frame 1: a
Frame 2: null
Frame 3: null
Frame 4: null
Frame 5: null

Sequence Number: 1
Reference: s
Page Fault: true
Frame 1: a
Frame 2: s
Frame 3: null
Frame 4: null
Frame 5: null

Sequence Number: 2
Reference: d
Page Fault: true
Frame 1: a
Frame 2: s
Frame 3: d
Frame 4: null
Frame 5: null

Sequence Number: 3
Reference: f
Page Fault: true
Frame 1: a
Frame 2: s
Frame 3: d
Frame 4: f
Frame 5: null

Sequence Number: 4
Reference: r
Page Fault: true
Frame 1: r
Frame 2: s
Frame 3: d
Frame 4: f
Frame 5: null

Sequence Number: 5
Reference: e
Page Fault: true
Frame 1: e
Frame 2: s
Frame 3: d
Frame 4: f
Frame 5: null

Total page faults: 6
```

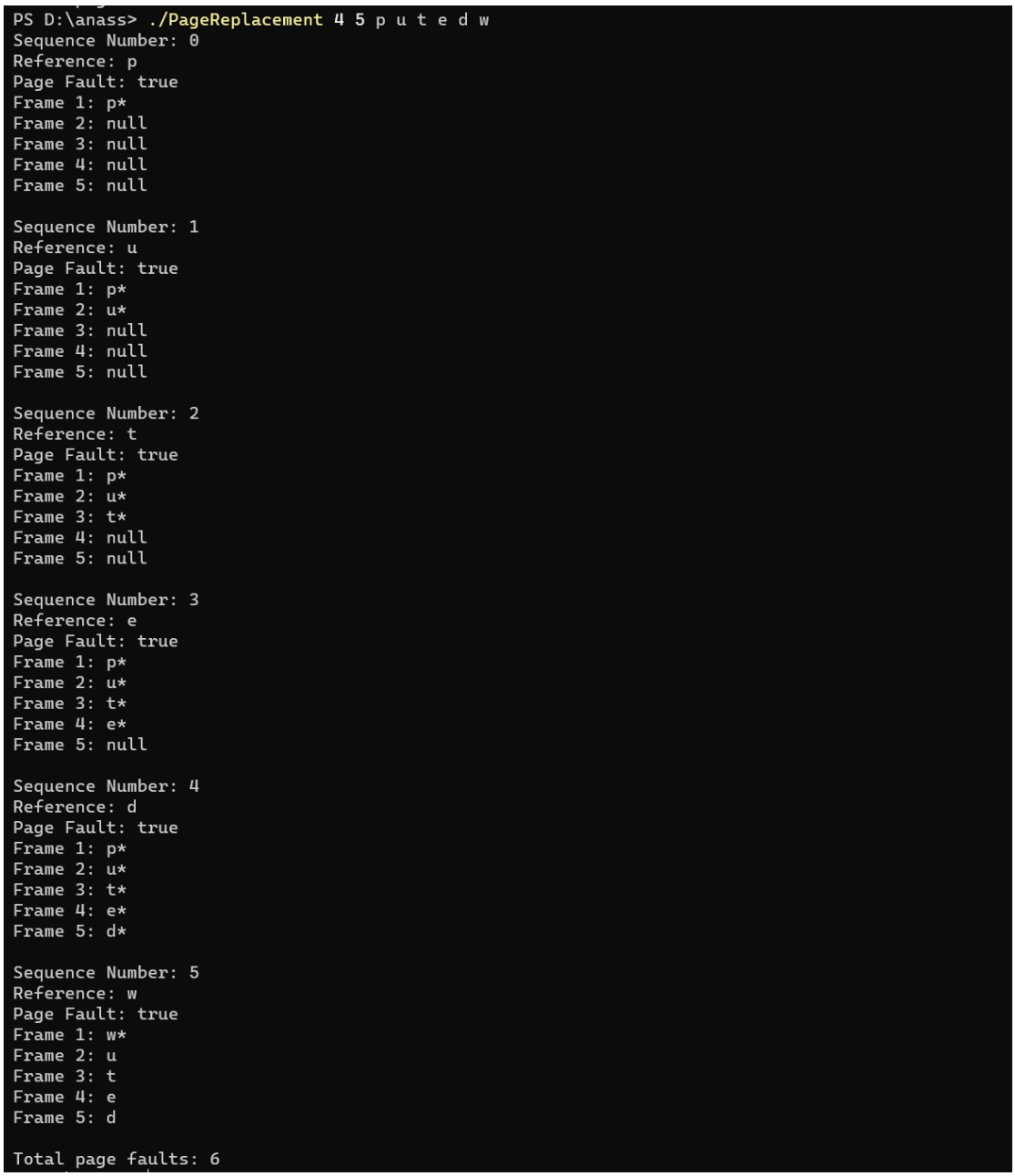
- Sample 2:

```
PS D:\anass> ./PageReplacement 1 5 a s d f r e
Sequence Number: -11153225
Reference: a
Page Fault: true
Frame 1: a (victim)
Frame 2: null
Frame 3: null
Frame 4: null
Frame 5: null

Sequence Number: -11153225
Reference: s
Page Fault: true
Frame 1: a (victim)
Frame 2: s
Frame 3: null
Frame 4: null
Frame 5: null

Sequence Number: -11153225
Reference: d
Page Fault: true
Frame 1: a (victim)
Frame 2: s
Frame 3: d
Frame 4: null
Frame 5: null

Sequence Number: -11153225
Reference: f
Page Fault: true
Frame 1: a (victim)
Frame 2: s
Frame 3: d
Frame 4: f
Frame 5: null

Sequence Number: -11153225
Reference: r
Page Fault: true
Frame 1: a (victim)
Frame 2: s
Frame 3: d
Frame 4: f
Frame 5: r

Sequence Number: -11153225
Reference: e
Page Fault: true
Frame 1: e
Frame 2: s (victim)
Frame 3: d
Frame 4: f
Frame 5: r

Total page faults: 6
```

- Sample 3:

```
PS D:\anass> ./PageReplacement 2 5 h o d u r e
Sequence Number: 0
Reference: h
Page Fault: true
Frame 1: h
Frame 2: null
Frame 3: null
Frame 4: null
Frame 5: null

Sequence Number: 1
Reference: o
Page Fault: true
Frame 1: o
Frame 2: h
Frame 3: null
Frame 4: null
Frame 5: null

Sequence Number: 2
Reference: d
Page Fault: true
Frame 1: d
Frame 2: o
Frame 3: h
Frame 4: null
Frame 5: null

Sequence Number: 3
Reference: u
Page Fault: true
Frame 1: u
Frame 2: d
Frame 3: o
Frame 4: h
Frame 5: null

Sequence Number: 4
Reference: r
Page Fault: true
Frame 1: r
Frame 2: u
Frame 3: d
Frame 4: o
Frame 5: h

Sequence Number: 5
Reference: e
Page Fault: true
Frame 1: e
Frame 2: r
Frame 3: u
Frame 4: d
Frame 5: o

Total page faults: 6
PS D:\anass>
```

- Sample 4:




# 5 Analysis and Conclusion

### Page Faults and Their Occurrence

Page faults occur when a program attempts to access data that is not currently in the main memory (RAM). In the context of our Page Replacement Algorithms program, a page fault is counted each time the system tries to access a page that is not in the set of available frames. The frequency of these faults depends on the specific algorithm used and the pattern of page references.

- **FIFO:** Page faults occur frequently with older pages, regardless of their recent usage. This can lead to unnecessary replacements if older pages are still being used frequently.
- **LRU:** Page faults are less frequent compared to FIFO, as LRU considers recent usage. However, it might not always be optimal, as it doesn't consider the frequency of page usage.
- **Optimal:** Theoretically, this algorithm results in the least number of page faults, as it always replaces the page that will not be needed for the longest duration.
- **Clock:** A practical and efficient version of LRU, offering a good balance between performance and complexity. Page faults occur less frequently, but like LRU, it doesn't account for the frequency of page usage.

**Comparison of Different Algorithms**

- **FIFO** is simple but can lead to unnecessary replacements, known as Belady's anomaly.
- **LRU** is more efficient than FIFO as it considers recent page usage, but it requires more complex data tracking.
- **Optimal** provides the best possible strategy in terms of minimizing page faults but is impractical for real-world use due to its requirement for future knowledge.
- **Clock** offers a practical and efficient approach, balancing between FIFO and LRU's characteristics but without their major drawbacks.

**Future Improvements**

In the future, the program can be enhanced by:

- **Implementing More Algorithms:** Including more sophisticated algorithms like LFU (Least Frequently Used) or MFU (Most Frequently Used).
- **User Interface:** Developing a user-friendly GUI to make the program more accessible to non-technical users.
- **Performance Optimization**: Enhancing the efficiency of current algorithms, especially for larger datasets.
- **Real-time Analysis:** Incorporating features that allow real-time monitoring of page faults and algorithm performance.

Learnings from the Project

- **Algorithm Efficiency:** Understanding how different algorithms perform under various scenarios and their impact on system performance.
- **Data Structure Relevance:** Learning the significance of choosing the right data structure for solving specific problems.
- **System Design:** Gaining insights into system design and modular programming, enhancing readability and maintainability of code.
- **Practical Implementation:** Realizing the difference between theoretical algorithm performance and practical application, particularly in the context of memory management.

This project provided valuable insights into memory management and the inner workings of operating systems, emphasizing the importance of algorithm selection based on specific use cases and system requirements.