VIDHI ROHIRA

S.Y B.TECH

SEM III

COMPUTER ENGINEERING

OS LAB 6

231071052

BATCH – C

LABORATORY 6

AIM:- Implement page replacement algorithms.

THEORY:-

There are mainly three page replacement algorithms that we study and implement:
1] First In First Out (FIFO).
2] Least Frequently Used (LFU).
3] Least Recently Used (LRU).
4] Optimal Page Replacement.

KEY CONCEPTS:-

1. Pages and Frames

- **Pages:** In a virtual memory system, programs are divided into fixed-size blocks known as pages.
- **Frames:** These are fixed-size blocks of physical memory where pages are stored once loaded. The number of frames is limited by the available physical memory.

2. Page Fault

- **Definition:** A page fault occurs when a process tries to access a page not currently in memory.
- **Action:** The operating system loads the required page from disk into memory, potentially swapping out an existing page to free up space if needed.

3. Page Hit

- A **page hit** occurs when a process tries to access a page (a block of memory) that is already loaded in the physical memory. In other words, if the requested page is found in memory, it is a "hit," meaning there is no need to fetch the page from disk.

- **Page Hit:** The page is already in memory.
- **Page Fault:** The page is not in memory and must be loaded from disk, possibly replacing an existing page in memory if space is needed.

1] FIRST IN FIRST OUT (FIFO):-

First In First Out (FIFO) is a page replacement algorithm in which the oldest page in memory is removed first when a new page needs to be loaded. This approach uses a queue to track the order of pages, with the earliest loaded page being the first to be removed.

Advantages of FIFO

1. **Simplicity:** FIFO is easy to understand and implement as it only requires tracking the order in which pages were loaded.
2. **Predictable Behavior:** It removes the oldest page consistently, making the behavior predictable.

Disadvantages of FIFO

1. **Poor Performance in Some Cases:** FIFO can suffer from Belady's Anomaly, where adding more frames increases page faults.
2. **No Usage Awareness:** It does not consider how frequently or recently pages are accessed, which can lead to removing pages that are still in use.
3. **Inefficiency:** FIFO may evict a frequently used page that was loaded earlier, resulting in higher page faults and potentially lower performance compared to other algorithms.

ALGORITHM:-

FIFO Replacement

- **Mechanism**: The FIFO algorithm replaces the page that has been in memory the longest.
- **Assumption**: It operates under the assumption that pages that have been in memory the longest are less likely to be needed soon.

1.] Initialize:

- Create an empty queue to keep track of pages in memory.
- Initialize an empty set to store pages currently in memory.
- Set the page fault counter to zero.

2.] For each page in the page reference string:

- If the page is already in memory (page hit):
  - Continue to the next page (no action required).
- If the page is not in memory (page fault):
  - Increment the page fault counter.
  - If memory (frame) is full:
    - Remove the oldest page (front of the queue) from the set and the queue.
  - Add the new page to the set and to the back of the queue (FIFO order).

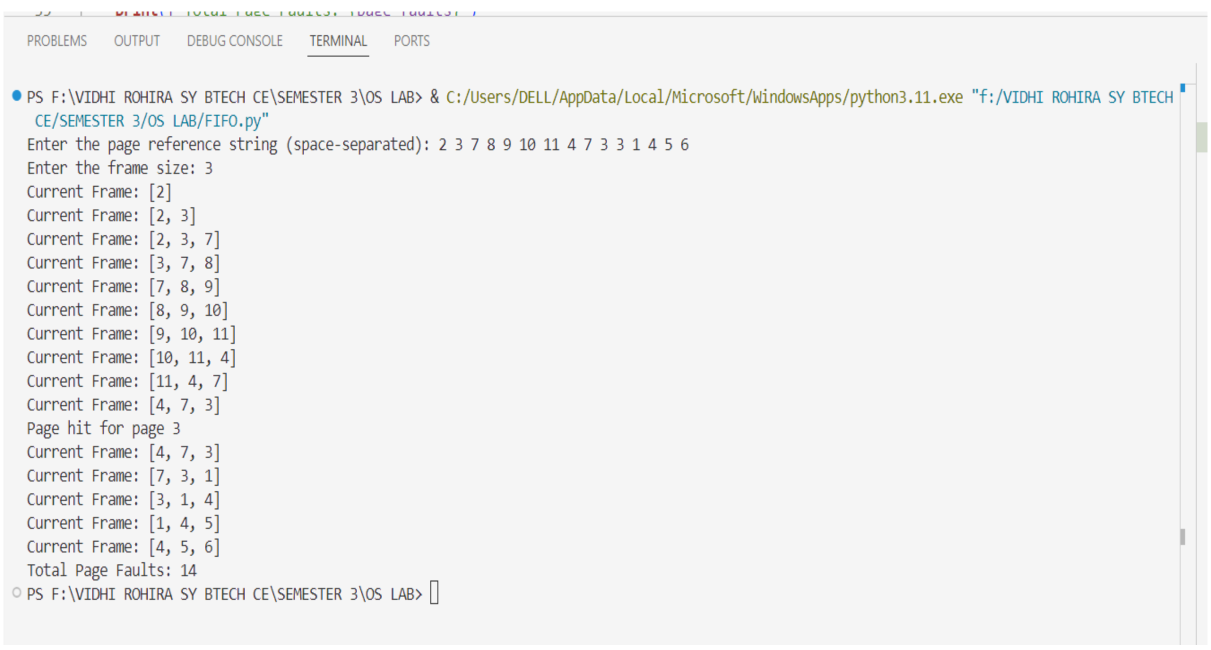
3.] End of algorithm:

- The page fault counter will reflect the total number of page faults.
- The queue will contain the current state of memory (pages in FIFO order).

CODE:

```python
from collections import deque

def fifo_algo(page_requests, frame_size):
    # Check if frame size is valid
    if frame_size <= 0:
        print("Invalid frame size!")
        return

    # Initialize necessary data structures
    page_set = set()  # Set to store pages in memory (for fast lookups)
    page_order = deque()  # Queue to maintain the order of pages (FIFO)
    page_faults = 0  # Counter for page faults

    # Iterate over each page in the page reference list
    for page in page_requests:
        # Page hit
        if page in page_set:
            print(f"Page hit for page {page}")
        else:
            # Page fault
            page_faults += 1
            if len(page_set) >= frame_size:
                # Remove the oldest page (FIFO)
                old_page = page_order.popleft()
                page_set.remove(old_page)

            # Add the new page to the set and queue
            page_set.add(page)
            page_order.append(page)

        # Display the current state of memory (page frame)
        print("Current Frame:", list(page_order))

    # Print the total number of page faults
    print(f"Total Page Faults: {page_faults}")

# Main function to get user input and call fifo_algo
if __name__ == "__main__":
    # Input for page reference string
    page_requests = list(map(int, input("Enter the page reference string
(space-separated): ").split()))
    frame_size = int(input("Enter the frame size: "))

    # Call FIFO page replacement function
    fifo_algo(page_requests, frame_size)
```

OUTPUT:



```
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB> & C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.11.exe "f:/VIDHI ROHIRA SY BTECH
 CE/SEMESTER 3/OS LAB/FIFO.py"
Enter the page reference string (space-separated): 2 3 7 8 9 10 11 4 7 3 3 1 4 5 6
Enter the frame size: 3
Current Frame: [2]
Current Frame: [2, 3]
Current Frame: [2, 3, 7]
Current Frame: [3, 7, 8]
Current Frame: [7, 8, 9]
Current Frame: [8, 9, 10]
Current Frame: [9, 10, 11]
Current Frame: [10, 11, 4]
Current Frame: [11, 4, 7]
Current Frame: [4, 7, 3]
Page hit for page 3
Current Frame: [4, 7, 3]
Current Frame: [7, 3, 1]
Current Frame: [3, 1, 4]
Current Frame: [1, 4, 5]
Current Frame: [4, 5, 6]
Total Page Faults: 14
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB>
```

2] LEAST FREQUENTLY USED (LFU):-

Definition: The LFU page replacement algorithm removes the page that has been accessed the least number of times. It assumes that pages accessed infrequently in the past are likely to be used less frequently in the future.

How it Works:

- Each page in memory has a counter that tracks the number of times it has been accessed.
- When a page needs to be replaced, the page with the lowest access count (frequency) is chosen.
- If multiple pages have the same frequency, ties are typically broken by choosing the oldest page (or using FIFO for those pages).

Advantages of LFU

1. **Usage Awareness**: LFU considers how often a page is accessed, which helps retain frequently used pages in memory.
2. **Improved Performance in Certain Scenarios**: For workloads with predictable access patterns, LFU can reduce page faults compared to simpler algorithms like FIFO.

Disadvantages of LFU

1. **Complexity**: Tracking the frequency of each page adds overhead, requiring additional data structures, which can slow down the system.
2. **Aging Issue**: LFU can retain pages that were frequently accessed in the past but are no longer needed ("aging").
3. **Not Always Optimal**: LFU is not effective for workloads with sudden changes in access patterns, as it may retain pages that are no longer in active use.

ALGORITHM:

A. LFU Replacement

- **Mechanism**: The LFU algorithm replaces the page that has been accessed the least number of times.
- **Assumption**: It operates under the assumption that pages that have been used less frequently are less likely to be needed in the near future.

B. Frequency Counter

- **Function**: LFU uses a frequency counter to track the number of times each page has been accessed.
- **Replacement**: When a page needs to be replaced, the algorithm selects the page with the lowest access frequency.

---

1. Initialize:

- Create an empty list **page_frame** to store pages currently in memory.
- Create an empty dictionary **frequency** to track the frequency of access for each page.
- Set **page_faults** to 0.

2. For each page in the page reference string:

- If the page is already in the memory (Page Hit):
  - Increment the frequency counter for the page.
- If the page is not in the memory (Page Fault):
  - Increment the page fault counter.
  - If the frame is full:
    - Find the page with the lowest frequency in the memory.
    - If multiple pages have the same frequency, choose one arbitrarily (or based on additional criteria, such as the oldest page).
    - Remove the page with the lowest frequency from memory.
  - Add the new page to the memory (insert it into **page_frame**).
  - Initialize the frequency of the new page to 1.

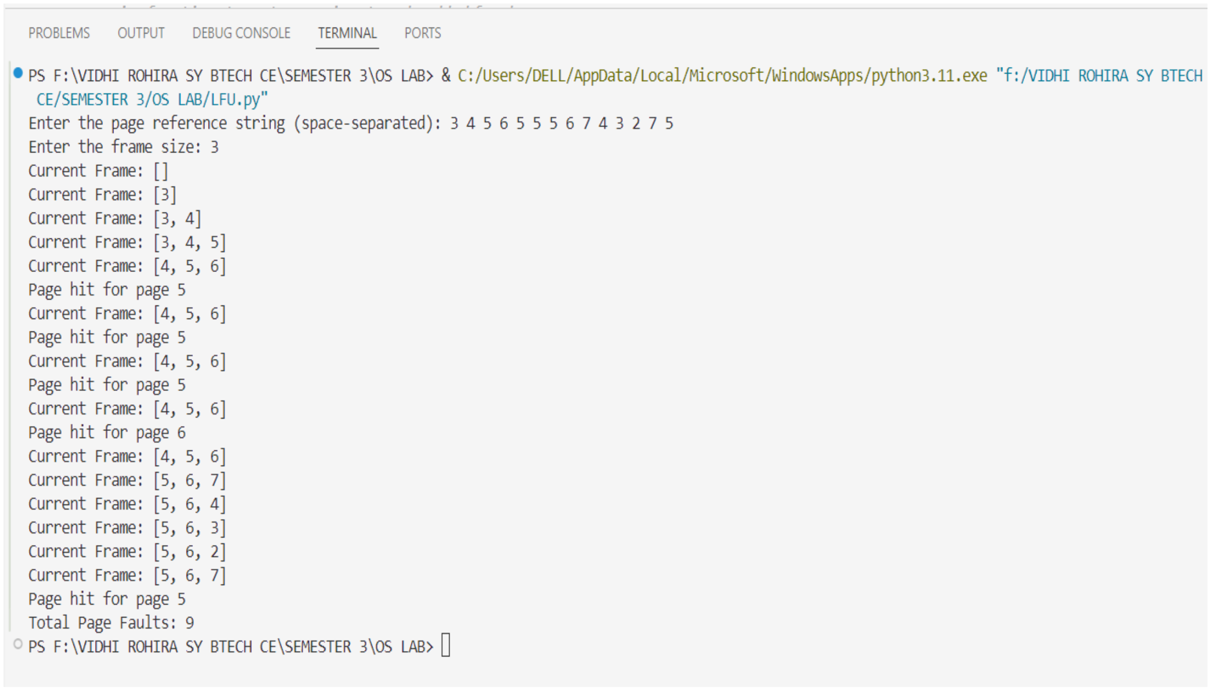
3. End of Algorithm:

- Print the total number of page faults.

CODE:

```python
from collections import defaultdict

def lfu_algo(page_requests, frame_size):
    page_frame = []  # List to store pages in memory
    frequency = defaultdict(int)  # Dictionary to store the frequency of each 
page
    page_faults = 0

    for page in page_requests:
        # Display current frame before processing the page
        print("Current Frame:", page_frame)

        # Check if page is already in the frame (Page Hit)
        if page in page_frame:
            print(f"Page hit for page {page}")
            frequency[page] += 1  # Increase the frequency of the page
        else:
            # Page Fault occurs (page not found)
            page_faults += 1

            # If the frame is full, remove the least frequently used page
            if len(page_frame) >= frame_size:
                # Find the least frequently used page
                lfu_page = page_frame[0]
                min_freq = frequency[lfu_page]

                # Find the page with the least frequency
                for p in page_frame:
                    if frequency[p] < min_freq:
                        lfu_page = p
                        min_freq = frequency[p]

                # Remove the least frequently used page
                page_frame.remove(lfu_page)
                del frequency[lfu_page]  # Remove its frequency entry

            # Insert the new page
            page_frame.append(page)
            frequency[page] = 1  # Initialize its frequency to 1

    # Display the total number of page faults
    print(f"Total Page Faults: {page_faults}")


# Main function to get user input and call lfu_algo
if __name__ == "__main__":
    # Input for page reference string
    page_requests = list(map(int, input("Enter the page reference string 
(space-separated): ").split()))
    frame_size = int(input("Enter the frame size: "))

    # Call LFU page replacement function
    lfu_algo(page_requests, frame_size)
```

OUTPUT:-


PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB> & C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.11.exe "f:/VIDHI ROHIRA SY BTECH
 CE/SEMESTER 3/OS LAB/LFU.py"
Enter the page reference string (space-separated): 3 4 5 6 5 5 5 6 7 4 3 2 7 5
Enter the frame size: 3
Current Frame: []
Current Frame: [3]
Current Frame: [3, 4]
Current Frame: [3, 4, 5]
Current Frame: [4, 5, 6]
Page hit for page 5
Current Frame: [4, 5, 6]
Page hit for page 5
Current Frame: [4, 5, 6]
Page hit for page 5
Current Frame: [4, 5, 6]
Page hit for page 6
Current Frame: [4, 5, 6]
Current Frame: [5, 6, 7]
Current Frame: [5, 6, 4]
Current Frame: [5, 6, 3]
Current Frame: [5, 6, 2]
Current Frame: [5, 6, 7]
Page hit for page 5
Total Page Faults: 9
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB> 
```

3.] LEAST RECENTLY USED:

Definition: The **LRU** page replacement algorithm replaces the page that has not been used for the longest period of time. It assumes that pages which have been most recently used are more likely to be needed again soon, while older pages are less likely to be reused in the near future.

How LRU Works:

- Every time a page is accessed, it is marked as recently used.
- When a page needs to be replaced (because the frame is full), the page that hasn't been used for the longest time is chosen for replacement.

Advantages of LRU:

1. **More Efficient than FIFO**: LRU typically performs better than FIFO because it takes into account the recency of access, making it more likely to keep frequently accessed pages in memory.
2. **Realistic Assumption**: The idea behind LRU (that recently accessed pages are more likely to be accessed again) aligns with real-world usage patterns.
3. **Simple to Understand and Implement**: LRU is intuitive and relatively straightforward to implement with the right data structures (e.g., a queue or linked list).

Disadvantages of LRU:

1. **Overhead for Implementation**: Keeping track of the order in which pages are accessed can require additional resources, such as maintaining a list or a more complex data structure (like a stack or deque), which can be costly in terms of time and space.
2. **Not Optimal for All Workloads**: For certain types of workloads, where page access is not based on recency, LRU may not be as effective as other algorithms like LFU or FIFO.
3. **Cache Thrashing**: In certain scenarios (like alternating between two pages), LRU can lead to cache thrashing, where pages are continuously swapped in and out of memory, leading to poor performance.

ALGORITHM:

A. LRU Replacement:

- **Mechanism**: The LRU algorithm replaces the page that has not been used for the longest time. Pages that have been accessed more recently are assumed to be used again soon, and thus remain in memory.

B. Tracking Page Access:

- **Tracking Method**: LRU requires tracking the order of page accesses. Typically, this is done using a queue or linked list where the least recently used pages are at the end of the list, and the most recently used pages are at the front.

C. Page Queue:

- **Queue/Linked List**: A queue or list is used to track the order in which pages are accessed. When a page is accessed, it moves to the front of the queue. The least recently used page is evicted when space is needed for a new page.

1.] Initialize:

- Create an empty list page_frame to store pages currently in memory (the current frame).
- Initialize page_faults to 0 to count the number of page faults.

2.] For each page in the page reference string:

- **Check if the page is already in the frame (Page Hit)**:
    - If the page is found in page_frame, it is a page hit.
    - Move this page to the end of the list to mark it as the most recently used (this ensures that the least recently used page is always at the front).
- **If the page is not in the frame (Page Fault)**:
    - Increment the page fault counter.
    - **If the frame is full (i.e., the number of pages in page_frame is equal to the frame size)**:
        - Remove the least recently used page (i.e., remove the first page in the list).
    - Add the new page to the list, marking it as the most recently used.

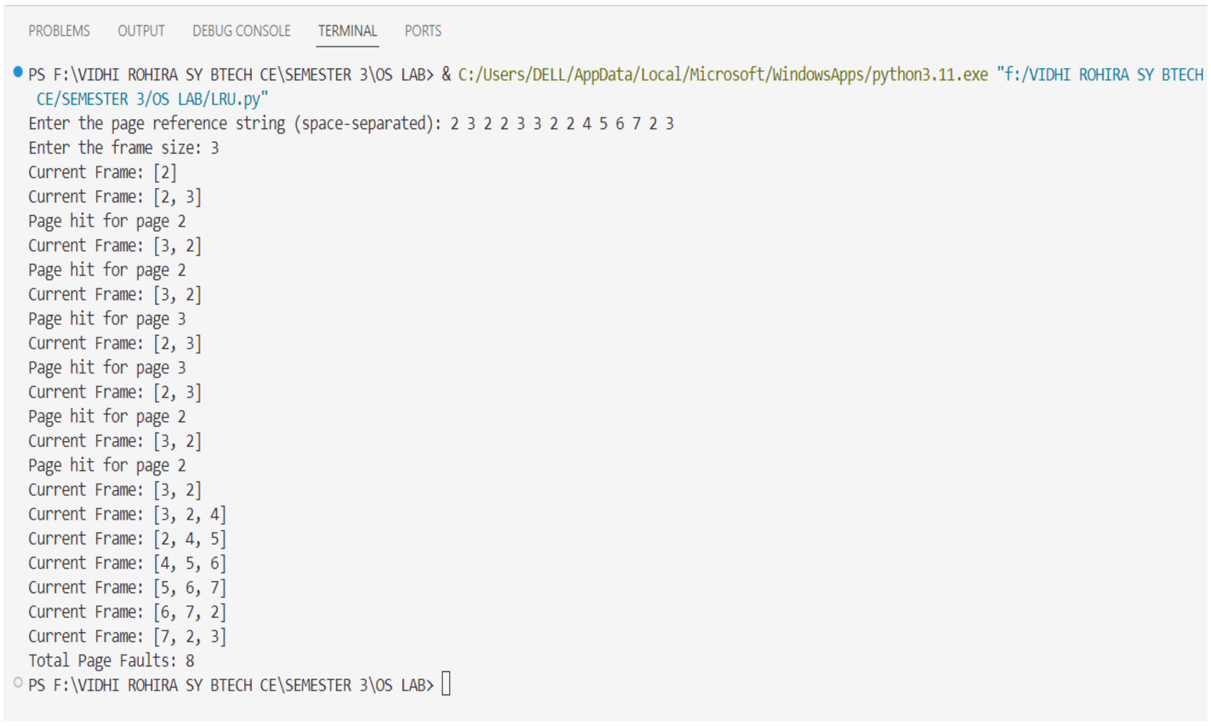
3.] End of Algorithm:

- After processing all pages, display the total number of page faults.

CODE:-

```python
def lru_algo(pages, frame_size):
    page_frame = []  # List to store pages in memory
    page_faults = 0  # Counter for page faults

    # Iterate through each page in the reference string
    for page in pages:
        # Check if the page is already in the frame (page hit)
        if page in page_frame:
            print(f"Page hit for page {page}")

            # Move this page to the end of the list to mark it as most
recently used
            page_frame.remove(page)
            page_frame.append(page)
        else:
            # Page Fault: The page is not in the frame
            page_faults += 1

            # If frame is full, remove the least recently used (LRU) page
            if len(page_frame) >= frame_size:
                page_frame.pop(0)  # Remove the LRU page (first in the list)

            # Add the new page as the most recently used
            page_frame.append(page)

        # Display the current state of the page frame
        print("Current Frame:", page_frame)

    # Display the total number of page faults after processing all pages
    print(f"Total Page Faults: {page_faults}")


# Main function to get user input
if __name__ == "__main__":
    # Input for page reference string
    pages = list(map(int, input("Enter the page reference string (space-
separated): ").split()))
    frame_size = int(input("Enter the frame size: "))

    # Call the LRU page replacement algorithm function
    lru_algo(pages, frame_size)
```

OUTPUT:-


PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB> & C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.11.exe "f:/VIDHI ROHIRA SY BTECH
 CE/SEMESTER 3/OS LAB/LRU.py"
Enter the page reference string (space-separated): 2 3 2 2 3 3 2 2 4 5 6 7 2 3
Enter the frame size: 3
Current Frame: [2]
Current Frame: [2, 3]
Page hit for page 2
Current Frame: [3, 2]
Page hit for page 2
Current Frame: [3, 2]
Page hit for page 3
Current Frame: [2, 3]
Page hit for page 3
Current Frame: [2, 3]
Page hit for page 2
Current Frame: [3, 2]
Page hit for page 2
Current Frame: [3, 2]
Current Frame: [3, 2, 4]
Current Frame: [2, 4, 5]
Current Frame: [4, 5, 6]
Current Frame: [5, 6, 7]
Current Frame: [6, 7, 2]
Current Frame: [7, 2, 3]
Total Page Faults: 8
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB>
```

4] OPTIMAL PAGE REPLACEMENT:

The **Optimal Page Replacement Algorithm** (also known as **OPT**) is a page replacement algorithm that aims to minimize the number of page faults by replacing the page that will not be used for the longest period of time in the future. This algorithm is theoretical, as it requires knowledge of future page references, which is typically not available in practice. However, it serves as a benchmark for evaluating other page replacement algorithms.

How it Works:

1. **When a page needs to be loaded into memory**:
    - If the page is already in memory, it is a **page hit**, and no replacement is needed.
    - If the page is not in memory, it is a **page fault**, and the page to replace is selected based on the following rule:
        - From the pages in memory, choose the one that will be used farthest in the future or will not be used at all. This page is replaced with the new page.
2. **Page Fault Handling**:
    - If there is space in memory, simply load the new page.
    - If memory is full, replace the page that has the farthest next use or is never used again.

Advantages:

1. **Optimal Performance**: This algorithm guarantees the minimum number of page faults, making it the most efficient page replacement algorithm in terms of minimizing page faults.
2. **Benchmark**: It serves as a theoretical upper bound for the performance of all other page replacement algorithms.

Disadvantages:

1. **Future Knowledge Requirement**: The algorithm requires knowledge of future page references, which is impossible to have in real-world scenarios.
2. **Complexity**: It is computationally expensive to find the page that will be used farthest in the future.
3. **Not Practical for Real Systems**: Since it cannot predict the future, it is not used in real operating systems but provides a benchmark for comparison with other algorithms.

CONCLUSION:-

In this lab session, we studied and implemented three page replacement algorithms: FIFO (First In, First Out), LRU (Least Recently Used), and LFU (Least Frequently Used). FIFO replaces the oldest page, which is simple but inefficient. LRU replaces the least recently used page, making it more effective in many scenarios, while LFU replaces the least frequently used page. We also explored the Optimal Page Replacement (OPT) algorithm, which is theoretically the best for minimizing page faults, as it replaces the page that will not be used for the longest time. However, OPT cannot be implemented in practice because it requires knowledge of future page references. Among the algorithms implemented, LRU and LFU are more practical for real systems, while OPT serves as a benchmark for evaluating other algorithms.