PAGE REPLACEMENT ALGORITHM

Experiment No: 6 **Date: / /23**

Aim: a) To implement First In First Out Page replacement algorithm.

Theory:

The FIFO page replacement algorithm is one of the simplest and most straightforward algorithms used in operating systems to manage the pages in a page table. The main idea behind FIFO is to replace the oldest page in memory when a page fault occurs.

How FIFO Works:

Page Frame Queue: Maintain a queue that represents the order in which pages were brought into memory. The oldest page is at the front of the queue, and the newest page is at the rear.

Page Request: When a page is requested, check if it is in the page table (in memory).

Page Hit: If the requested page is already in memory (a page hit), no further action is needed. The page is simply accessed.

Page Fault (Page Miss): If the requested page is not in memory (a page fault), the operating system must bring the page into a page frame.

Page Replacement: If all page frames are occupied, select the page at the front of the queue (the oldest page) for replacement.

Update Page Frame Queue: Remove the oldest page from the front of the queue and add the new page to the rear.

Update Page Table: Update the page table to reflect the new page in the page frame.

Advantages:

- Simple to implement.
- Requires minimal bookkeeping.

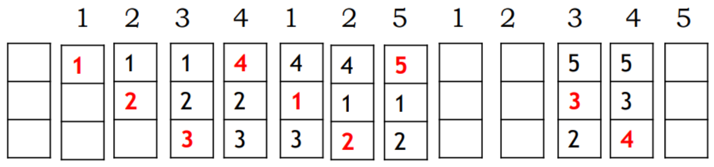
Disadvantages:

- Does not consider the access frequency or importance of pages.
- May suffer from the Belady's anomaly, where increasing the number of page frames may result in more page faults.

Example:

Reference string: 1,2,3,4,1,2,5,1,2,3,4,5

3 frames (3 pages can be in memory at a time per process)



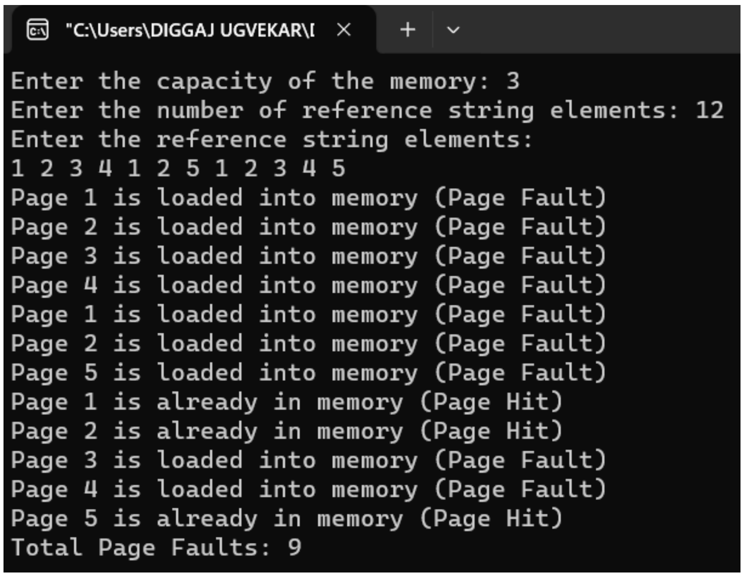
	1	2	3	4	1	2	5	1	2	3	4	5
	1	1	1	4	4	4	5			5	5	
		2	2	2	1	1	1			3	3	
			3	3	3	2	2			2	4	

Total Page faults = 9

Program

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_set>
using namespace std;
void fifoPageReplacement(const vector<int>& referenceString, int capacity) {
    int pageFaults = 0;
    queue<int> pageQueue;
    unordered_set<int> pageSet;
    for (int i = 0; i < referenceString.size(); ++i) {
        int currentPage = referenceString[i];
        // Check if the page is in the set (page hit)
        if (pageSet.find(currentPage) != pageSet.end()) {
            cout << "Page " << currentPage << " is already in memory (Page Hit)\n";
        } else {
            // Page fault: page is not in memory
            ++pageFaults;
            // Check if the pageQueue is full
            if (pageQueue.size() == capacity) {
                int oldestPage = pageQueue.front();
                pageQueue.pop();
                pageSet.erase(oldestPage);
            }
            // Add the current page to the pageQueue and set
            pageQueue.push(currentPage);
            pageSet.insert(currentPage);
           cout << "Page " << currentPage << " is loaded into memory (Page Fault)\n";
        }}
    cout << "Total Page Faults: " << pageFaults << endl;
}
int main() {
    int capacity;
    cout << "Enter the capacity of the memory: ";
    cin >> capacity;
    int n;
    cout << "Enter the number of reference string elements: ";
    cin >> n;
    vector<int> referenceString(n);
    cout << "Enter the reference string elements:\n";
    for (int i = 0; i < n; ++i) {
        cin >> referenceString[i]; }
    fifoPageReplacement(referenceString, capacity);
    return 0;
}
```

Output



```
Enter the capacity of the memory: 3
Enter the number of reference string elements: 12
Enter the reference string elements:
1 2 3 4 1 2 5 1 2 3 4 5
Page 1 is loaded into memory (Page Fault)
Page 2 is loaded into memory (Page Fault)
Page 3 is loaded into memory (Page Fault)
Page 4 is loaded into memory (Page Fault)
Page 1 is loaded into memory (Page Fault)
Page 2 is loaded into memory (Page Fault)
Page 5 is loaded into memory (Page Fault)
Page 1 is already in memory (Page Hit)
Page 2 is already in memory (Page Hit)
Page 3 is loaded into memory (Page Fault)
Page 4 is loaded into memory (Page Fault)
Page 5 is already in memory (Page Hit)
Total Page Faults: 9
```

Conclusion

First In First Out Page replacement algorithm was studied and implemented successfully.

PAGE REPLACEMENT ALGORITHM

Experiment No: 6 **Date: / /23**

Aim: b) To implement optimal page replacement algorithm.

Theory:

The Optimal Page Replacement Algorithm, also known as the MIN (Minimum) algorithm, is an optimal algorithm for page replacement in the context of virtual memory management. The basic idea behind this algorithm is to replace the page that will not be used for the longest period of time in the future.

Advantages and Disadvantages:

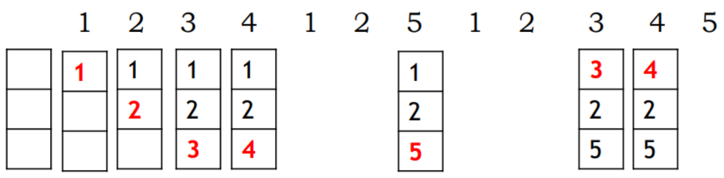
The Optimal Page Replacement Algorithm provides the lowest possible page-fault rate since it makes the optimal replacement decision based on future references.

However, it is practically impossible to implement in a real system due to the requirement of knowing future references.

Example:

Reference string: 1,2,3,4,1,2,5,1,2,3,4,5

3 frames (3 pages can be in memory at a time per process)



Total Page faults = 7

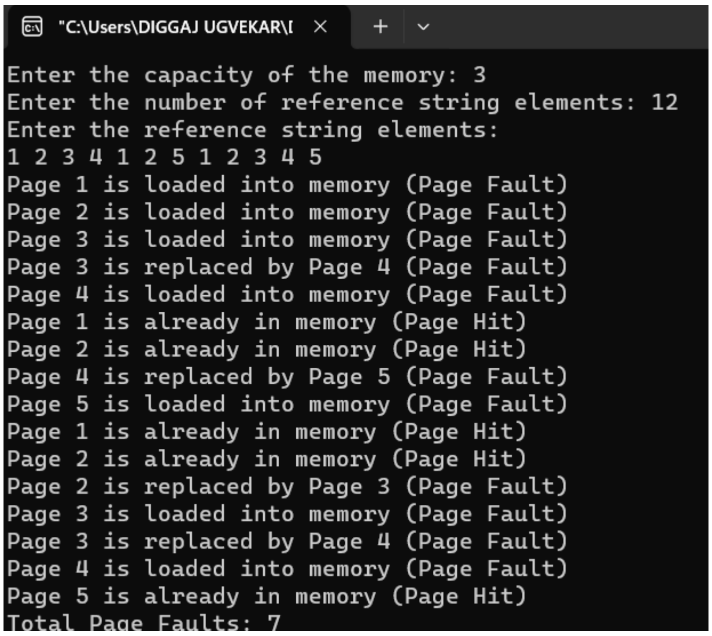
Program

```
#include <iostream>

#include <vector>

#include <unordered_map>

#include <algorithm>

#include <climits>

using namespace std;

int optimalPageReplacement(const vector<int>& referenceString, int capacity) {

unordered_map<int, int> pageNextUse;  // Maps page numbers to their next use index
```

```
int pageFaults = 0;
for (int i = 0; i < referenceString.size(); ++i) {
int currentPage = referenceString[i];
if (pageNextUse.find(currentPage) != pageNextUse.end()) {
cout << "Page " << currentPage << " is already in memory (Page Hit)\n";
} else {
++pageFaults;
if (pageNextUse.size() == capacity) {
int pageToReplace = -1;
int furthestNextUse = -1;
for (const auto& entry : pageNextUse) {
int nextUseIndex = entry.second;
auto futureUseIt = find(referenceString.begin() + i, referenceString.end(), entry.first);
if (futureUseIt == referenceString.end()) {
pageToReplace = entry.first;
break;
}
int distanceToNextUse = distance(referenceString.begin() + i, futureUseIt);
if (distanceToNextUse > furthestNextUse) {
furthestNextUse = distanceToNextUse;
pageToReplace = entry.first;
}}
// Remove the page with the furthest next use
pageNextUse.erase(pageToReplace);
cout << "Page " << pageToReplace << " is replaced by Page " << currentPage << " (Page
Fault)\n";
}
auto nextUseIt = find(referenceString.begin() + i, referenceString.end(), currentPage);
pageNextUse[currentPage] = (nextUseIt == referenceString.end()) ? INT_MAX :
distance(referenceString.begin(), nextUseIt);
cout << "Page " << currentPage << " is loaded into memory (Page Fault)\n";}}
```

```
return pageFaults;}
int main() {
int capacity;
cout << "Enter the capacity of the memory: ";
cin >> capacity;
int n;
cout << "Enter the number of reference string elements: ";
cin >> n;
vector<int> referenceString(n);
cout << "Enter the reference string elements:\n";
for (int i = 0; i < n; ++i) {
cin >> referenceString[i];
}
int pageFaults = optimalPageReplacement(referenceString, capacity);
cout << "Total Page Faults: " << pageFaults << endl;
return 0;
}
```

Output



```
Enter the capacity of the memory: 3
Enter the number of reference string elements: 12
Enter the reference string elements:
1 2 3 4 1 2 5 1 2 3 4 5
Page 1 is loaded into memory (Page Fault)
Page 2 is loaded into memory (Page Fault)
Page 3 is loaded into memory (Page Fault)
Page 3 is replaced by Page 4 (Page Fault)
Page 4 is loaded into memory (Page Fault)
Page 1 is already in memory (Page Hit)
Page 2 is already in memory (Page Hit)
Page 4 is replaced by Page 5 (Page Fault)
Page 5 is loaded into memory (Page Fault)
Page 1 is already in memory (Page Hit)
Page 2 is already in memory (Page Hit)
Page 2 is replaced by Page 3 (Page Fault)
Page 3 is loaded into memory (Page Fault)
Page 3 is replaced by Page 4 (Page Fault)
Page 4 is loaded into memory (Page Fault)
Page 5 is already in memory (Page Hit)
Total Page Faults: 7
```

Conclusion

Optimal page replacement algorithm was studied and implemented successfully.

PAGE REPLACEMENT ALGORITHM

Experiment No: 6

Date: / /23

Aim: **c)** To implement Least Recently Used page replacement algorithm.

Theory:

The Least Recently Used (LRU) replacement algorithm aims to keep track of the order in which pages are accessed in memory and prioritize removing the page that hasn't been used for the longest time.

Here's a simple explanation of the LRU algorithm:

Tracking Page Usage:

LRU keeps track of the order in which pages are accessed. This can be implemented using a counter or a stack.

Page Access Update:

Every time a page is accessed, it is moved to the front of the list or given a lower counter value. This signifies that it has been used most recently.

Page Replacement Decision:

When a new page needs to be brought into memory and there is no space, LRU selects the page that has the highest counter value or is at the back of the list. This indicates that it hasn't been used for the longest time.

Implementation:

LRU can be implemented using a counter for each page or by maintaining a stack/queue of page numbers. When a page is used, it is moved to the front (or the top) of the stack, indicating its recent use.

Advantages and Disadvantages:

LRU performs well in capturing temporal locality by favoring pages that have been recently used.

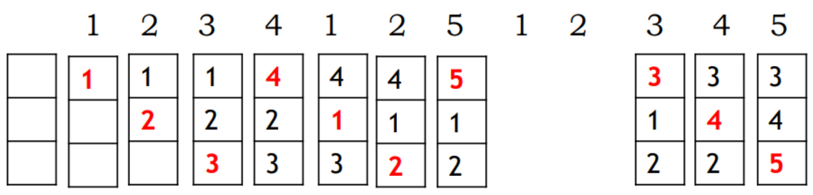
However, its implementation may involve maintaining a record of all page accesses, making it a bit more complex compared to some other algorithms.interrupted after its time quantum expires, and the CPU scheduler selects the next process to run.

Completion: The process continues to execute in a cyclic manner until all processes have completed their execution.

Example:

Reference string: 1,2,3,4,1,2,5,1,2,3,4,5

3 frames (3 pages can be in memory at a time per process)



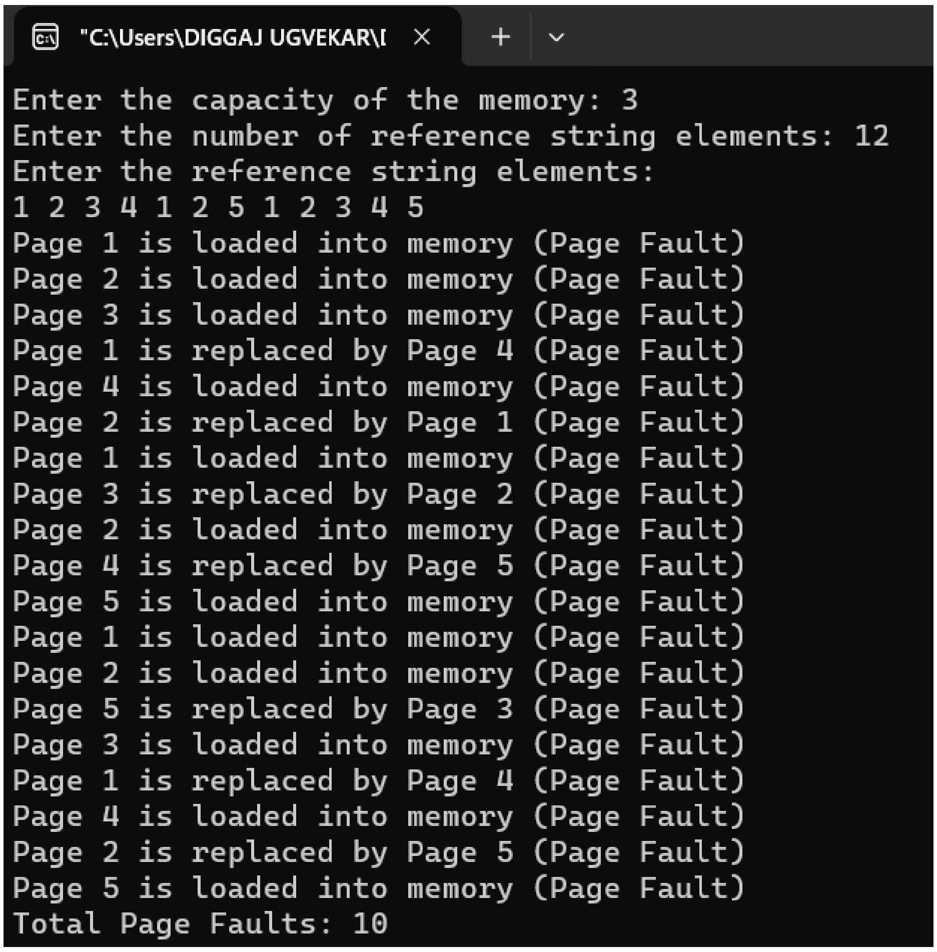
Total Page faults = 10

Program

```
#include <iostream>
#include <list>
#include <unordered_map>
using namespace std;
class LRUCache {
private:
    int capacity;
    list<int> recentList;
    unordered_map<int, list<int>::iterator> pageMap;
    int pageFaults;
public:
    LRUCache(int cap) : capacity(cap), pageFaults(0) {}
    void refer(int pageNum) {
        // If page is not in the cache
        if (pageMap.find(pageNum) == pageMap.end()) {
            // Check if the cache is full
            if (recentList.size() == capacity) {
                int leastRecent = recentList.back();
                recentList.pop_back();
                pageMap.erase(leastRecent);
cout << "Page " << leastRecent << " is replaced by Page " << pageNum << " (Page Fault)\n";
```

```
            }
            ++pageFaults;
        } else {
            // If the page is in the cache, remove it from the current position
            recentList.erase(pageMap[pageNum]);
        }
  recentList.push_front(pageNum);
        pageMap[pageNum] = recentList.begin();
        cout << "Page " << pageNum << " is loaded into memory (Page Fault)\n";
    }
    int getPageFaults() const {
        return pageFaults;
    }};
int main() {
    int capacity;
    cout << "Enter the capacity of the memory: ";
    cin >> capacity;

    LRUCache lruCache(capacity);

    int n;
    cout << "Enter the number of reference string elements: ";
    cin >> n;

    cout << "Enter the reference string elements:\n";
    for (int i = 0; i < n; ++i) {
        int pageNum;
        cin >> pageNum;
        lruCache.refer(pageNum);
    }
    cout << "Total Page Faults: " << lruCache.getPageFaults() << endl;


    return 0;
}
```

Output



```
"C:\Users\DIGGAJ UGVEKAR\I
Enter the capacity of the memory: 3
Enter the number of reference string elements: 12
Enter the reference string elements:
1 2 3 4 1 2 5 1 2 3 4 5
Page 1 is loaded into memory (Page Fault)
Page 2 is loaded into memory (Page Fault)
Page 3 is loaded into memory (Page Fault)
Page 1 is replaced by Page 4 (Page Fault)
Page 4 is loaded into memory (Page Fault)
Page 2 is replaced by Page 1 (Page Fault)
Page 1 is loaded into memory (Page Fault)
Page 3 is replaced by Page 2 (Page Fault)
Page 2 is loaded into memory (Page Fault)
Page 4 is replaced by Page 5 (Page Fault)
Page 5 is loaded into memory (Page Fault)
Page 1 is loaded into memory (Page Fault)
Page 2 is loaded into memory (Page Fault)
Page 5 is replaced by Page 3 (Page Fault)
Page 3 is loaded into memory (Page Fault)
Page 1 is replaced by Page 4 (Page Fault)
Page 4 is loaded into memory (Page Fault)
Page 2 is replaced by Page 5 (Page Fault)
Page 5 is loaded into memory (Page Fault)
Total Page Faults: 10
```

Conclusion

Least Recently Used page replacement algorithm was studied and implemented successfully.