Q-Write a program to implement Optimal page replacement algorithm.

```
#include <iostream>

using namespace std;


int search(int key, int frame_items[], int frame_occupied)

{

for (int i = 0; i < frame_occupied; i++)

if (frame_items[i] == key

return 1;

return 0;

}

void printOuterStructure(int max_frames)

{

printf("Stream ");

for (int i = 0; i < max_frames; i++)

printf("Frame%d ", i + 1);

}

void printCurrFrames(int item, int frame_items[], int frame_occupied, int max_frames)

{

printf("\n%d \t\t", item);

for (int i = 0; i < max_frames; i++)

   {

if (i < frame_occupied)

printf("%d \t\t", frame_items[i]);

else

printf("- \t\t");

   }

}

int predict(int ref_str[], int frame_items[], int refStrLen, int index, int frame_occupied)

{

int result = -1, farthest = index;

for (int i = 0; i < frame_occupied; i++)

   {
```

```
int j;
for (j = index; j < refStrLen; j++)
      {
if (frame_items[i] == ref_str[j])
         {
if (j > farthest)
            {
farthest = j;
result = i;
            }
break;
         }
      }
if (j == refStrLen)
return i;
   }
return (result == -1) ? 0 : result;
}

void optimalPage(int ref_str[], int refStrLen, int frame_items[], int max_frames)
{
int frame_occupied = 0;
printOuterStructure(max_frames);
int hits = 0;
for (int i = 0; i < refStrLen; i++)
   {
if (search(ref_str[i], frame_items, frame_occupied))
      {
hits++;
printCurrFrames(ref_str[i], frame_items, frame_occupied, max_frames);
continue;
}
if (frame_occupied < max_frames)
      {
         frame_items[frame_occupied] = ref_str[i];
```

```
frame_occupied++;
printCurrFrames(ref_str[i], frame_items, frame_occupied, max_frames);
      }
else
      {
int pos = predict(ref_str, frame_items, refStrLen, i + 1, frame_occupied);
          frame_items[pos] = ref_str[i];
printCurrFrames(ref_str[i], frame_items, frame_occupied, max_frames);
      }
   }
printf("\n\nHits: %d\n", hits);
printf("Misses: %d", refStrLen - hits);
}

int main()
{
int ref_str[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};
int refStrLen = sizeof(ref_str) / sizeof(ref_str[0]);
int max_frames = 3;
int frame_items[max_frames];
optimalPage(ref_str, refStrLen, frame_items, max_frames);
return 0;
}
```

Q-Write a program to implement Least Recently Used (LRU) page replacement algorithm.

```
#include <iostream>
#include <unordered_map>
#include <list>

using namespace std;

class LRUCache {
public:
LRUCache(int capacity) : _capacity(capacity) {}
```

```
int get(int key) {
auto it = _cache.find(key);
if (it == _cache.end()) return -1;

        // Move accessed page to the front of the list
        _lru.splice(_lru.begin(), _lru, it->second);

return it->second->second;
    }

void put(int key, int value) {
auto it = _cache.find(key);
if (it != _cache.end()) {
            // Update the value and move the page to the front of the list
it->second->second = value;
            _lru.splice(_lru.begin(), _lru, it->second);
return;
        }

if (_cache.size() >= _capacity) {
            // Remove the least recently used page from the cache
int lruKey = _lru.back().first;
            _cache.erase(lruKey);
            _lru.pop_back();
        }

        // Add the new page to the cache and the front of the list
        _lru.emplace_front(key, value);
        _cache[key] = _lru.begin();
    }

private:
int _capacity;
list<pair<int, int>> _lru; // List to keep track of the access order
```

```
    unordered_map<int, list<pair<int, int>>::iterator> _cache; // Map to quickly look up a page
};

int main() {
    LRUCache cache(2);

cache.put(1, 1);
cache.put(2, 2);
cout<< cache.get(1) << endl; // Returns 1
cache.put(3, 3); // Evicts key 2
cout<< cache.get(2) << endl; // Returns -1 (not found)
cache.put(4, 4); // Evicts key 1
cout<< cache.get(1) << endl; // Returns -1 (not found)
cout<< cache.get(3) << endl; // Returns 3
cout<< cache.get(4) << endl; // Returns 4

return 0;
}
```

Q-Write a program to implement First In First Out (FIFO) page replacement algorithm.

```
// C++ implementation of FIFO page replacement
// in Operating Systems.
#include<bits/stdc++.h>
using namespace std;

// Function to find page faults using FIFO
int pageFaults(int pages[], int n, int capacity)
{
        // To represent set of current pages. We use
        // an unordered_set so that we quickly check
        // if a page is present in set or not
        unordered_set<int> s;

        // To store the pages in FIFO manner
        queue<int> indexes;

        // Start from initial page
        int page_faults = 0;
        for (int i=0; i<n; i++)
        {
                // Check if the set can hold more pages
                if (s.size() < capacity)
                {
                        // Insert it into set if not present
                        // already which represents page fault
                        if (s.find(pages[i])==s.end())
                        {
                                // Insert the current page into the set
```

```
                                        s.insert(pages[i]);

                                        // increment page fault
                                        page_faults++;

                                        // Push the current page into the queue
                                        indexes.push(pages[i]);
                                }
                        }

                        // If the set is full then need to perform FIFO
                        // i.e. remove the first page of the queue from
                        // set and queue both and insert the current page
                        else
                        {
                                // Check if current page is not already
                                // present in the set
                                if (s.find(pages[i]) == s.end())
                                {
                                        // Store the first page in the
                                        // queue to be used to find and
                                        // erase the page from the set
                                        int val = indexes.front();

                                        // Pop the first page from the queue
                                        indexes.pop();

                                        // Remove the indexes page from the set
                                        s.erase(val);

                                        // insert the current page in the set
                                        s.insert(pages[i]);

                                        // push the current page into
                                        // the queue
                                        indexes.push(pages[i]);

                                        // Increment page faults
                                        page_faults++;
                                }
                        }
                }

                return page_faults;
}

// Driver code
int main()
{
                int pages[] = {7, 0, 1, 2, 0, 3, 0, 4,
                                                2, 3, 0, 3, 2};
                int n = sizeof(pages)/sizeof(pages[0]);
                int capacity = 4;
                cout<< pageFaults(pages, n, capacity);

return0;
}
```