First In First Out (FIFO)

In operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

**First In First Out (FIFO) page replacement algorithm –**  
This is the simplest page replacement algorithm. In this algorithm, operating system keeps track of all pages in the memory in a queue, oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

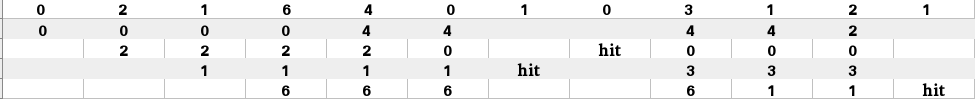
**Example -1.** Consider page reference string 1, 3, 0, 3, 5, 6 and 3 page slots.?

Initially all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> 3 **Page Faults.**  
when 3 comes, it is already in memory so —> 0 Page Faults.  
Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —>1 **Page Fault.**  
Finally 6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 —>1 **Page Fault.**

So total page faults = **5**.

**Example -2.** Consider the following reference string: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1.

Using FIFO page replacement algorithm –



So, total number of page faults = 9.

Given memory capacity (as number of pages it can hold) and a string representing pages to be referred, write a function to find number of page faults.

**Implementation –** Let capacity be the number of pages that memory can hold. Let set be the current set of pages in memory.

|  |
| --- |
| 1- Start traversing the pages.  i) If set holds less pages than capacity.  a) Insert page into the set one by one until the size of set reaches capacity or all  page requests are processed.  b) Simultaneously maintain the pages in the queue to perform FIFO.  c) Increment page fault  ii) Else  If current page is present in set, do nothing.  Else  a) Remove the first page from the queue as it was the first to be entered in the memory  b) Replace the first page in the queue with the current page in the string.  c) Store current page in the queue.  d) Increment page faults.  2. Return page faults. |

**Program:**

|  |
| --- |
| // Java implementation of FIFO page replacement  // in Operating Systems.  import java.util.HashSet;  import java.util.LinkedList;  import java.util.Queue;  class Test  {  // Method to find page faults using FIFO  static 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  HashSet<Integer> s = new HashSet<>(capacity);    // To store the pages in FIFO manner  Queue<Integer> indexes = new LinkedList<>() ;    // 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.contains(pages[i]))  {  s.add(pages[i]);    // increment page fault  page\_faults++;    // Push the current page into the queue  indexes.add(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.contains(pages[i]))  {  //Pop the first page from the queue  int val = indexes.peek();    indexes.poll();    // Remove the indexes page  s.remove(val);    // insert the current page  s.add(pages[i]);    // push the current page into  // the queue  indexes.add(pages[i]);    // Increment page faults  page\_faults++;  }  }  }    return page\_faults;  }    // Driver method  public static void main(String args[])  {  int pages[] = {7, 0, 1, 2, 0, 3, 0, 4,  2, 3, 0, 3, 2};  int capacity = 4;  System.out.println(pageFaults(pages, pages.length, capacity));  }  } |

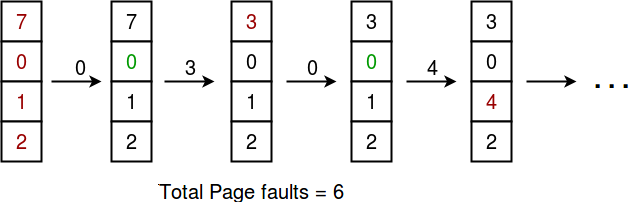
**Belady’s anomaly –**  
Belady’s anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. For example, if we consider reference string 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4 and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10 page faults.

Least Recently Used (LRU)

In operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

In **Least Recently Used (LRU)** algorithm is a Greedy algorithm where the page to be replaced is least recently used. The idea is based on locality of reference, the least recently used page is not likely   
Let say the page reference string ***7 0 1 2 0 3 0 4 2 3 0 3 2*** . Initially we have ***4 page*** slots empty.

Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —>**4 Page faults**   
0 is already their so —> **0 Page fault.**   
when 3 came it will take the place of 7 because it is least recently used —>**1 Page fault**   
0 is already in memory so —>**0 Page fault**.   
4 will takes place of 1 —>**1 Page Fault**   
Now for the further page reference string —>**0 Page fault** because they are already available in the memory.



|  |
| --- |
| **Note :**  **Given memory capacity (as number of pages it can hold) and a string representing pages to be referred, write a function to find number of page faults.** |
| **Algorithm :**  Let **capacity** be the number of pages that  memory can hold. Let **set** be the current  set of pages in memory.  1- Start traversing the pages.  i) **If set holds less pages than capacity.**  a) Insert page into the set one by one until  the size of **set** reaches **capacity** or all  page requests are processed.  b) Simultaneously maintain the recent occurred  index of each page in a map called **indexes**.  c) Increment page fault  ii) **Else**  **If** current page is present in **set**, do nothing.  **Else**  a) Find the page in the set that was least  recently used. We find it using index array.  We basically need to replace the page with  minimum index.  b) Replace the found page with current page.  c) Increment page faults.  d) Update index of current page.  2. Return page faults. |

|  |
| --- |
| // Java program for page replacement algorithms  import java.util.ArrayList;  public class LRU {    // Driver method  public static void main(String[] args) {  int capacity = 4;  int arr[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};    // To represent set of current pages.We use  // an Arraylist  ArrayList<Integer> s=new ArrayList<>(capacity);  int count=0;  int page\_faults=0;  for(int i:arr)  {  // Insert it into set if not present  // already which represents page fault  if(!s.contains(i))  {    // Check if the set can hold equal pages  if(s.size()==capacity)  {  s.remove(0);  s.add(capacity-1,i);  }  else  s.add(count,i);  // Increment page faults  page\_faults++;  ++count;    }  else  {  // Remove the indexes page  s.remove((Object)i);  // insert the current page  s.add(s.size(),i);  }    }  System.out.println(page\_faults);  }  } |