**­­LRU CACHE IMPLEMENTATION**

**LRU CACHE IMPLEMENTATION:**

The Least Recently Used (LRU) page replacement policy **replaces the page that has not been used for the longest period of time** The optimal algorithm assumes the entire reference string to be present at the time of allocation and replaces the page that will not be used for the longest period of time. LRU page replacement policy is based on the observation that pages that have been heavily used in the last few instructions will probably be heavily used again in the next few. Conversely, pages that have not been used for ages will probably remain unused for a long time.

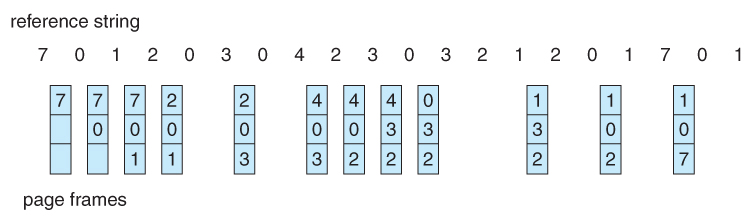
We are also given cache (or memory) size (Number of page frames that cache can hold at a time). The LRU caching scheme is to remove the least recently used frame when the cache is full and a new page is referenced which is not there in cache.

**Page Miss**

If the requested page is not found in the main memory ( physical memory ) it is referred as Page Miss.

**Page Hit**

If the requested page is found in the main memory ( physical memory ) it is referred as Page Hit.

example:

* + - Here the No. of Frames=3
    - Initially there are no pages in the Frame.
    - It requests the first page ( i.e 7) , it looks the buffer it is available or not. If it is not available then the requested page is brought into the main memory such a page is denoted as **Page Miss.**
    - Suppose if the requested page is available in main memory it is denoted as **Page Hit.**
    - This process will happen until all the frames are the full. (Here 7, 0 ,1 are filled)
    - After all frames are filled .It requests another page ( Here 2), it looks if the requested page is available in the frame or not. If it is available it just update the recently used count.Here, it is not available at the same time the frames also full. We need to make a room for the req.page . so remove the page that has been least recently used and add the req.page to the frame.
    - This process will continue until all the requested pages has been completed through LRU.

**Approach for O(1):**

1. Queue which is implemented using a doubly linked list. The maximum size of the queue will be equal to the total number of frames available (cache size). The most recently used pages will be near front end and least recently pages will be near the rear end.
2. A Hash with page number as key and address of the corresponding queue node as value.

When a page is referenced, the required page may be in the memory. If it is in the memory, we need to detach the node of the list and bring it to the front of queue. If the required page is not in memory, we bring that in memory. In simple words, we add a new node to the front of the queue and update the corresponding node address in the hash. If the queue is full, i.e. all the frames are full, we remove a node from the rear of the queue, and add the new node to the front of the queue.

**Solution:**

1. Create node structure which has page number and pointer to previous and next node.
2. Create queue structure which has number of frames and pointer to front and rear end.
3. Create hash structure which has hash table which determine whether the page is in queue or not and pointer to the queue.
4. int nopages(struct node \*front)

Gets the front end of the queue and returns the number of nodes present in the queue. It is done using while loop by traversing nodes which stops when node is null.

1. struct Queue \*delete(struct Queue \*queue)

This function deletes the rear end of the queue. The queue is given as input. It returns the queue after deletion. The node previous to the rear is made as current rear.

1. struct Hash \*addnode(struct Hash \*hashq, int pageno)

This function adds an node(pageno) at front of the hash queue which is given as input. And returns the hash queue after adding that node. When no of node ( noofpages()) is less than frame size, adds the node at the front. Otherwise when equal deletes ( delete()) the node present at rear end.

1. struct Hash\* lru(struct Hash \*hashq, int pageno)

This function performs page replacement and returns the hash queue after replacement. This checks the hash table of that pageno given as input.

* 1. When it is null, it adds ( addnode()) the node ( pageno) at front of the queue and increments miss count.
  2. When it is not null and not equal to front end, it cuts that node from the queue and add it at the front end by giving the front end to that nodes next and make it as front and increment hit count.
  3. When it is not null and equal to front end, just increments the hit count.

1. Get the frame size, hash table size and file name which has reference string. And reads the input pageno from that file and performs replacement and prints the frame until the file reaches the end. After prints the hit count and miss count.