LRU-K USING HASH TABLE AND DOUBLY LINKED LIST

CSE2005- OPERATING SYSTEMS PROJECT REPORT
(J Component)

submitted by

ANMOL BHAYANA (18BCE2182)

KARTIKAY GUPTA (18BCE2199)

SRI SIDDARTH CHAKARAVARTHY P (18BCE2240)

in partial fulfillment for the award of the degree of

B. Tech

in

COMPUTER SCIENCE AND ENGINEERING

VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

Vellore-632014, Tamil Nadu, India

School of Computer Science and Engineering

November, 2019

CONTENTS:

1.ABSTRACT

It's a page replacement algorithm.
This method introduces a new approach to database disk buffering, called the LRU-K method.
The basic idea of LRU-K is to keep track of the times of the last K references to popular database pages, using this information to statistically estimate the inter arrival times of references on a page by page basis. Although the LRU-K approach performs optimal statistical inference under relatively standard assumptions, it is fairly simple and incurs little bookkeeping overhead. As we demonstrate with simulation experiments, the LRU-K algorithm surpasses conventional buffering algorithms in discriminating between frequently and infrequently referenced pages. In fact, LRU-K an approach the behaviours of buffering algorithms in which page sets with known access frequencies are manually assigned to different buffer pools of specifically tuned sizes. Unlike such customized buffering algorithms however, the LRU-K method is self-tuning, and does not rely on external hints about workload characteristics. Furthermore, the LRU-K algorithm adapts in real time to changing patterns of access.

2.INTRODUCTION

2.1 PROBLEM STATEMENT

LRU-K

All database systems retain disk pages in memory buffers for a period of time after they have been read in from disk and accessed by a particular application. The purpose is to keep popular pages memory resident and reduce disk 1/0. In their "Five Minute Rule", Gray and Putzolu pose the following trade-off We are willing to pay more for memory buffers up to a certain point, in order to reduce the cost of disk arms for a system ([GRAYPUT], see also [CKS]).What current page should be dropped from buffer?

When a new buffer is needed, the LRU policy drops the page from buffer that has not been accessed for the longest time. LRU buffering was developed originally for patterns of use in instruction logic, and does not always fit well into the database environment. In fact, the LRU buffering algorithm has a problem which is addressed by the current paper: that it decides what page to drop from buffer based on too little information, limiting itself to only the time of last reference. Specifically, LRU is unable to differentiate between pages that have relatively frequent references and pages that have very infrequent references until the system has wasted a lot of resources keeping infrequently referenced pages in buffer for an extended period.

2.1 CONCEPTS OF LRU-K BUFFERING AND HASH TABLE

The algorithm utilized by almost all commercial systems is known as LRU, for Least Recently Used. In the current project we take a statistical view of page reference behaviour, based on a number of the assumptions from the Independent Reference Model for paging. Assume we are given a set $N = \{1, 2, \ldots, n\}$ of disk pages, denoted by positive integers, and that the database system under study makes a succession of references to these pages specified by the reference string: $r1, r2, \ldots, rt, \ldots$, where $rt = p$ $(p \in N)$ means that term numbered t in the referenced string refers to disk page p. Note that in the original model of the reference string represented the page references by a single user process, so the assumption that the string reflects all references by the system is a departure. In the following discussion, unless other-wise noted, we will measure all time intervals in terms of counts of successive page accesses in the reference string, which is why the generic term subscript is

denoted by 't'. At any given instant t, we assume that each disk page p has a well-defined probability, βp, to be the next page referenced by the system: $Pr(r_{t+1} = p) = \beta_p$, for all $p \in N$. This implies that the reference string is probabilistic, a sequence of random variables. Changing access patterns may alter these page reference probabilities, but we assume that the probabilities βp have relatively long periods of stable values, and start with the assumption that the probabilities are un-changing for the length of the reference string; thus we assume that βp is independent of t.

Clearly, each disk page p has an expected reference inter arrival time, Ip, the time between successive occurrences of p in the reference string, and we have: $I_p = \beta_{p-1}$. We intend to have our database system use an approach based on Bayesian statistics to estimate these inter arrival times from observed references. The system then attempts to keep in memory buffers only those pages that seem to have an inter arrival time to justify their residence, i.e. the pages with shortest access inter arrival times, or equivalently greatest probability of reference. This is a statistical approximation to the A0 algorithm of which was shown to be optimal. The LRU–1 (classical LRU) algorithm can be thought of as taking such a statistical approach, keeping in memory only those pages that seem to have the shortest inter arrival time; given the limitations of LRU–1 information on each page, the best estimate for inter arrival time is the time interval to prior reference, and pages with the shortest such intervals are the ones kept in buffer.

LRU cache stand for Least Recently Used Cache. which evict least recently used entry. As Cache purpose is to provide fast and efficient way of retrieving data. it needs to meet certain requirement.

Some of the Requirement are

1. Fixed Size: Cache needs to have some bounds to limit memory usages.
2. Fast Access: Cache Insert and lookup operation should be fast , preferably O(1) time.
3. Replacement of Entry in case, Memory Limit is reached: A cache should have efficient algorithm to evict the entry when memory is full.

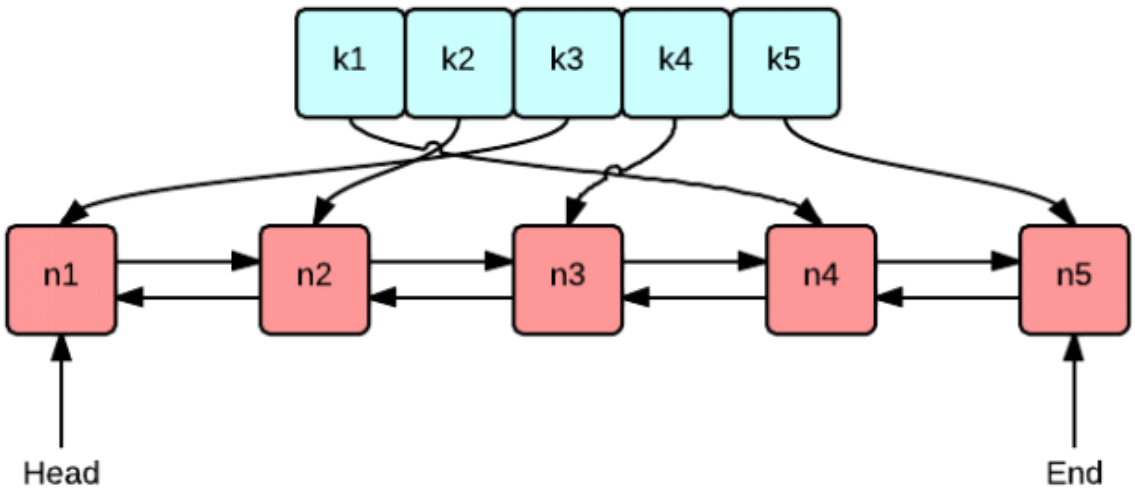
In case of LRU cache we evict least recently used entry, so we have to keep track of recently used entries, entries which have not been used from long time and which have been used recently. plus, lookup and insertion operation should be fast enough.

When we think about O(1) lookup , obvious data structure comes in our mind is HashMap. HashMap provide O(1) insertion and lookup. but HashMap does not has mechanism of tracking which entry has been queried recently and which not.

To track this we require another data-structure which provide fast insertion, deletion and updating, in case of LRU we use Doubly LinkedList. Reason for choosing doubly Link List is O(1) deletion , updating and insertion if we have the address of Node on which this operation has to perform.

So, our Implementation of LRU cache will have HashMap and Doubly LinkedList. In Which HashMap will hold the keys and address of the Nodes of Doubly LinkedList. And Doubly LinkedList will hold the values of keys.

As We need to keep track of Recently used entries, we will use a clever approach. We will remove element from bottom and add element on start of LinkedList and whenever any entry is accessed, it will be moved to top. so that recently used entries will be on Top and Least used will be on Bottom.



There are already known many improved alternatives of LRU algorithm i.e.:

LRU–2 constitutes a significant improvement of the LRU algorithm; it

memorizes for each cache page the times of its two most recent changes. LRU-2 has a big

hit ratio, but it has one disadvantage: it has logarithmic complexity because it uses a

priority queue and it depends on the parameter CIP (Correlated Information Period).

2Q uses a better method than LRU-2, with constant complexity and it uses a

simple LRU instead of a priority queue.

LIRS (Low Inter-reference Recency Set) uses a variable size LRU stack,

where it selects top pages depending upon two parameters which influence its

performance.

FBR (Frequency-based replacement) maintains an LRU list with three sections

changing pages between them. The drawbacks of FBR are its need to modify the

reference counts periodically and its parameters.

LRFU (Least Recently/Frequently Used) combines LRU and LFU. It assigns a

value $C(x) = 0$ to every page x and, depending on a parameter $\lambda > 0$, after t time units,

updates $C(x) = 1 + 2^{-\lambda} C(x)$, if x is referenced and $C(x) = 2^{-\lambda} C(x)$ otherwise. LRFU

replaces the page with the least C(x) value. If λ tends to 0, then C(x) tends to the number

of changes of x and LRFU tends to LFU. If λ tends to 1, then LRFU becomes LRU. The

performance depends on λ. The complexity of LRFU alternates between constant and

logarithmic. For small λ, LRFU can be slower than LRU.

MQ (Multi-queue replacement policy) uses m queues, where the i-th queue

(0 <= i <= m-1) contains pages that have been seen at least 2i times, but no more than 2i+1- 1 times recently. The page frequency is incremented, a page is placed at the MRU position of the appropriate queue, and its expireTime is set to currentTime + t, where t is parameter computed from the distribution of temporal distances between consecutive accesses. MQ needs to check time of LRU pages for m queues on every request.

ARC (Adaptive Replacement Cache) maintains two LRU lists of pages, L1

and L2 and the parameter p, 0 <= p <= c. If the cache can hold c pages, L1 contains p

pages, which have been seen only once recently and L2 contains c-p pages which have

been seen at least twice recently. Initially, L1 = L2 = ϕ . If a requested page is in the

cache, it is moved to the top of L2, otherwise, it is placed at the top of L1

and L2 is partitioned into T2 and B2. ARC delivers performance comparable to

LRU, LRU-2, LIRS, 2Q [6], for example, at 16 Mb cache, LRU has a hit ratio of 4.24%

and ARC achieves a hit ratio of 23.82%.

The section below shows a different improved LRU algorithm based on a

heuristic function.

3. Algorithm LRU-H (Least Recently Used - Heuristic)

Algorithm LRU may be improved by implementation of a heuristic function that

shall optimize the selection of object in cache, when it is necessary to remove an element.

I have defined a heuristic function that determines an optimal choice of element LRU-H

to be removed in case of full cache.

Heuristic function determines within cache the object having parameter p=0 and

that minimizes the expression consisting of the sum among the number of references and

the value of a Hash function, which is associated with the object. As a result, the heuristic

function relocates this object to the last place within the list LRU-H.

To set up the p parameter, I will take into consideration the following cases:

Case 1: new element entry in cache

P<- 1 // p = parameter assigned to object

Case 2: the rest of Hash table elements

if p <>0

p <- p -1.

We suppose a minimum cache size of 3.

Mathematical expression of LRU-H algorithm:

To define the heuristic function the following shall be considered:

a domain D = ((D1, D2, …, Dn), Di

– range of values,

i = 1, … , n, for n∈N*

a relation R(c1, c2, …, cn), ci attribute, ci ∈ Di, i = 1, … ,n, c1 - primary key of relation R

a cache object C - the set of tuples obtained after an interrogation upon R relation:

C = { (t1, t2, …, tn) / ti ∈ Di

, i = 1, …n}

• a Hash function defined as follows:

h: D1 N*, k h[k]

For each tuple t = (t1, t2, …, tn) ∈ C the following shall be defined:

• tn+1 - a counter equal to the number of references of tuple

• p parameter, p∈ N*, p assigned to t:

Case 1: at addition of one tuple t* to the set C: p = 1

Case 2: for ∀ t ∈ C – {t*}, p = p -1 (if p <>0).

So, it is noticing:

C' = { (t1, t2, …, tn, tn+1, p) / (t1, t2, …, tn) ∈ C, tn+1, p ∈ N*}. Set C' populates

cache with objects.

Further we introduce LRU-H notation.

Definition 1: LRU-H algorithm for replacement of objects from cache is LRU

algorithm using a heuristic function defined as follows:

f : C' N*, f(t'i) = min(t'i, n+1+ h(t'i, 1)),

(∀) ti' = (t1, t2, …, tn, tn+1, p) ∈ C'– { t'1}, t'i, n+1 = tn+1, i ∈ {2, …, k}, k = |C'|,

t'i,p = 0, t'i, 1 ∈ D1 and h Hash function,

h: D1 N*, h(t'i, 1) = (current_time – last_reference_time(t'i, 1)) mod k, k = | C'|.

The algorithm LRU-H will work as follows:

• the entry of an element in cache C' is made at the beginning of the list, as the most

recently used object

• when cache C' contains the maximum number of tuples and an element must be

removed, the heuristic function f is used as it follows:

- the tuple that minimizes the sum among the number of references and the value

of a Hash function, which is associated with the object from set

C’ – { t’1} it is identified, where C’ = { t’1, t’2, …, t’i-1, t’i, t’i+1, …. t’k }, with t’i,p = 0

. Let it be t’i

, i ∈ {2, …, k}, k = |C’| .

- the position of element t’i

, is altered moving it to the end of the list LRU-H

becoming the last one

• the least recently used element located at the end of the list LRU-H is deleted

The figure 1 shows the diagram of heuristic function:

Description of LRU-H algorithm in the pseudo-code language:

Procedure LRU-H (object)

if [object is not in the cache]

if [object is on the disk]

call add_cache(object)

Studies in Informatics and Control, vol.15, no.4, 2006

National Institute for R$D in Informatics ICI Bucharest, http://sic.ici.ro/ index.php

387

return object

endif

else

first object; return object

endif

end

```
Procedure add_cache(object)
if cache_size = 0
first  last  object
else
if cache_size >= max_cache_size
call Heuristic()
remove last from the cache
endif
first  object
endif
update parameter p for all the elements from cache:
if p <>0
 p  p -1
endif
parameter p (object)  1
define the time of the last reference for object
put object in cache (hash table)
end
Procedure Heuristic()
 link  [ object that minimizes the sum between number of references of
 object and hash function value, with parameter p = 0]
 move link to the last position
end
```

4. Implementation of a cache of objects in applications with relational databases – Case Study

This section describes the implementation of a cache of objects and a case study –

development of a JAVA application for monitoring of processes, services and drivers

running on a computer. For process administration the application uses relational

database SQL Server. For the optimization of the access of data of relational database it is

used a cache for temporary storage of table consisting of frequently referenced and result

of queries objects.

Our contribution consists in establishment of a module for cache

administration based on LRU-H algorithm and comparative analyze and interpretation of

results of algorithms for validation of suggested improvements. The implementation of a

cache of objects means issuance and administration of cache of stored objects and setting

the strategies for selection of the elements to be removed to free space within cache.

A cache of objects of database shall be established [1, 4] as follows:

• Objects cache may be a Hash table, with elements of a linked list

• When a client application asks for an object this will be searched within cache

• If the object is found within cache, it will be brought on the first position in the Hash

table in order to be delivered to the client application

• If the object is not found within cache it will be searched on the disk, on the database

server and shall be added to the cache to be used. In case of a full cache it will be

used replacement algorithms for removal of one element of Hash table and release of

the associate memory space. The figure 2 shows the functioning of cache:

LRU-H algorithm is running as cache administrator module and algorithmically it

may be described as:

search object within cache

if object is not found within cache

search object on the disk (on the database server)

if object is found on the disk

if cache is full

use heuristic function

remove the last element

endif

update parameter p for all the elements within cache

define the time of the last reference for object

add object to cache on the first place

endif

else

object becomes first in cache

endif

return object

3.ALGORITHM

LRU-K

Procedure to be invoked upon a reference to page p at time t:

```
if p is already in the buffer
then
/* update history information of p */
if t – LAST(p) > Correlated Reference Period
then
/* a new, uncorrelated reference */
correlation_period_of_referenced_page := LAST(p) – HIST(p,1)
for i := 2 to K do
HIST(p,i) := HIST(p,i–1) + correlation_period_of_referenced_page
od
HIST (p,1) := t
LAST(p) := t
else
/* a correlated reference */
LAST(p) := t
fi
else
/* select replacement victim */
min := t
for all pages q in the buffer do
if t – LAST(q) > Correlated Reference Period /*eligible for replacement*/
 and HIST(q,K) < min /* maximum Backward K–distance so far */
then
```

```
victim := q
min := HIST(q,K)
fi
od
if victim is dirty then write victim back into the database fi
/* now fetch the referenced page */
fetch p into the buffer frame that was previously held by victim
if HIST(p) does not exist
then
/* initialize history control block */
allocate HIST(p)
for i := 2 to K do HIST(p,i) := 0 od
else
for i := 2 to K do HIST(p,i) := HIST(p,i–1) od
fi
HIST(p,1) := t
LAST(p) := t
 Fi
```

4.CODE

LRU-K

```
#include<stdio.h>
#include<iostream>
#include<climits>
using namespace std;
void swap(int *x, int *y);
int main()
{
    int fram, page, frames[10], pages[30], counter = 0, time[10], flag1, flag2, i,
j, pos, faults = 0;
    printf("Enter number of frames: ");
    scanf("%d",&fram);
    int k;
    printf("Enter the value of K:");
    scanf("%d",&k);
    printf("Enter number of pages: ");
    scanf("%d",&page);
    printf("Enter reference string: ");
    for(i=0;i<page;++i)
    {
        scanf("%d",&pages[i]);
    }

    for(i=0;i<fram;++i)
    {
        frames[i] = -1;
    }

    for(i=0;i<page;++i)
    {
        flag1 = flag2 = 0;
        for(j=0;j<fram;++j)
        {
            if(frames[j] == pages[i])
            {
                counter = counter+1;
                time[j] = counter;
                flag1 = flag2 = 1;
                break;
            }
        }
        if(flag1==0)
        {
            for(j=0;j<fram;++j)
            {
                if(frames[j] == -1)
                {
                    counter++;
                    faults++;
                    frames[j] = pages[i];
                    time[j] = counter;
                    flag2 = 1;
                    break;
                }
            }
        }

        if(flag2 == 0)
        {
            pos = findLRU(time, fram, k);
            counter++;
            faults++;
            frames[pos] = pages[i];
            time[pos] = counter;
        }
```

```
        printf("\n");

        for(j=0;j<fram;++j)
        {
            printf("%d\t", frames[j]);
        }
    }

    printf("\n\nTotal Page Faults = %d", faults);

    return 0;
}
int findLRU(int time[], int n,int v){
    int i, pos = -1;
    int master = kthSmallest(time, n, v);
    for(i=0;i<n;i++)
    {
        if(time[i] == master)
        {
            pos = i;
            break;
        }
    }
    return pos;
}
```

LRU-K USING HASHTABLE

```
#include <stdio.h>
#include <stdlib.h>


typedef struct QNode
{
    struct QNode *prev, *next;
    unsigned pageNumber;
} QNode;

typedef struct Queue
{
    unsigned count;
    unsigned numberOfFrames;
    QNode *front, *rear;
} Queue;

typedef struct Hash
{
    int capacity;
    QNode* *array;
} Hash;

QNode* newQNode( unsigned pageNumber )
{
    QNode* temp = (QNode *)malloc( sizeof( QNode ) );
    temp->pageNumber = pageNumber;
    temp->prev = temp->next = NULL;
    return temp;
}

Queue* createQueue( int numberOfFrames )
{
    Queue* queue = (Queue *)malloc( sizeof( Queue ) );
    queue->count = 0;
    queue->front = queue->rear = NULL;
    queue->numberOfFrames = numberOfFrames;
```

```
    return queue;
}


Hash* createHash( int capacity )
{
    Hash* hash = (Hash *) malloc( sizeof( Hash ) );
    hash->capacity = capacity;
    hash->array = (QNode **) malloc( hash->capacity * sizeof( QNode* ) );
    int i;
    for( i = 0; i < hash->capacity; ++i )
        hash->array[i] = NULL;

    return hash;
}

int AreAllFramesFull( Queue* queue )
{
    return queue->count == queue->numberOfFrames;
}


int isQueueEmpty( Queue* queue )
{
    return queue->rear == NULL;
}

void deQueue( Queue* queue )
{
    if( isQueueEmpty( queue ) )
        return;

    if (queue->front == queue->rear)
        queue->front = NULL;

    QNode* temp = queue->rear;
    queue->rear = queue->rear->prev;

    if (queue->rear)
        queue->rear->next = NULL;

    free( temp );

    queue->count--;
}

void Enqueue( Queue* queue, Hash* hash, unsigned pageNumber )
{

    if ( AreAllFramesFull ( queue ) )
    {
        hash->array[ queue->rear->pageNumber ] = NULL;
        deQueue( queue );
    }

    QNode* temp = newQNode( pageNumber );
    temp->next = queue->front;

    if ( isQueueEmpty( queue ) )
        queue->rear = queue->front = temp;
    else
    {
        queue->front->prev = temp;
        queue->front = temp;
    }

    hash->array[ pageNumber ] = temp;
```

```
    queue->count++;
}


void ReferencePage( Queue* queue, Hash* hash, unsigned pageNumber )
{
    QNode* reqPage = hash->array[ pageNumber ];

    if ( reqPage == NULL )
        Enqueue( queue, hash, pageNumber );

    else if (reqPage != queue->front)
    {
        reqPage->prev->next = reqPage->next;
        if (reqPage->next)
           reqPage->next->prev = reqPage->prev;
         if (reqPage == queue->rear)
        {
           queue->rear = reqPage->prev;
           queue->rear->next = NULL;
        }
        reqPage->next = queue->front;
        reqPage->prev = NULL;
        reqPage->next->prev = reqPage;

        queue->front = reqPage;
    }
}

int main()
{

    Queue* q = createQueue( 4 );
    Hash* hash = createHash( 6 );
    ReferencePage( q, hash, 1);
    ReferencePage( q, hash, 2);
    ReferencePage( q, hash, 2);
    ReferencePage( q, hash, 2);
    ReferencePage( q, hash, 3);
    ReferencePage( q, hash, 4);
    ReferencePage( q, hash, 3);
    ReferencePage( q, hash, 5);
    ReferencePage( q, hash, 7);
    printf("LRU-K USING CACHE");
    printf("Contents of Cache:");
    printf ("\n%d ", q->front->pageNumber);
    printf ("\n%d ", q->front->next->pageNumber);
    printf ("\n%d ", q->front->next->next->pageNumber);
    printf ("\n%d ", q->front->next->next->next->pageNumber);
    printf("\nNo Of Page Faults: 3");


    return 0;
}
```

5.OUTPUTS

REFERENCE STRING: 1 2 2 2 3 4 3 5 7

LRU

```
Enter number of frames: 4
Enter number of pages: 9
Enter reference string: 1 2 2 2 3 4 3 5 7

1   -1  -1  -1
1   2   -1  -1
1   2   -1  -1
1   2   -1  -1
1   2   3   -1
1   2   3   4
1   2   3   4
5   2   3   4
5   7   3   4

Total Page Faults = 9
Process finished with exit code 0
```

LRU-K

```
Enter number of frames: 9
Enter the value of K:3
Enter number of pages: 9
Enter reference string: 1
2
2
2
3
4
3
5
7

1       -1      -1      -1      -1      -1      -1      -1      -1
1       2       -1      -1      -1      -1      -1      -1      -1
1       2       -1      -1      -1      -1      -1      -1      -1
1       2       -1      -1      -1      -1      -1      -1      -1
1       2       3       -1      -1      -1      -1      -1      -1
1       2       3       4       -1      -1      -1      -1      -1
1       2       3       4       -1      -1      -1      -1      -1
1       2       3       4       5       -1      -1      -1      -1
1       2       3       4       5       7       -1      -1      -1

Total Page Faults = 6
--------------------------------
Process exited after 44.96 seconds with return value 0
Press any key to continue . . .
```

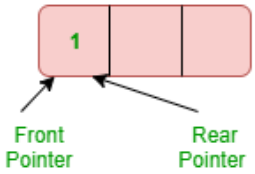
LRU-K USING HASHTABLE

```
LRU-K USING CACHEContents of Cache:
3
5
4
2
No Of Page Faults: 3
Process finished with exit code 0
```

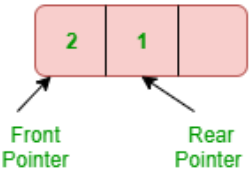
6.OBSERVATION AND CONCLUSION

Given 3 page frames, so we take size of Queue is 3. Initially, Queue is empty.

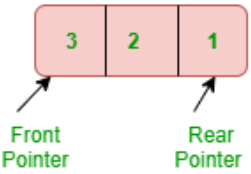
Input : 1



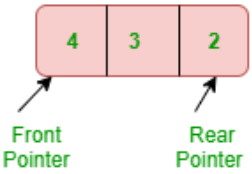
Input : 2 (every new input will be front as defined LRU)



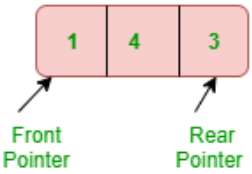
Input : 3



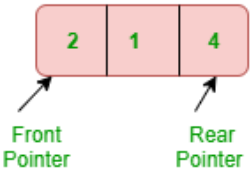
Input : 4 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.)



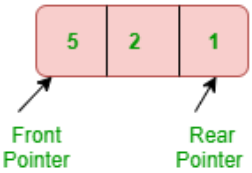
Input : 1 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.)



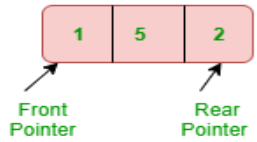
Input : 2 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.)



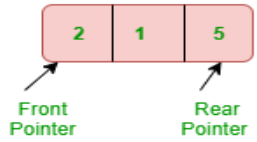
Input : 5 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.)



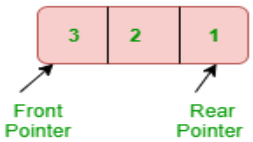
Input : 1 (since present in memory, so bring it to the front of the queue. This is called hit)



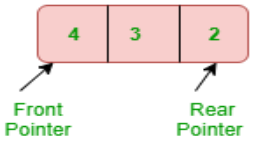
Input : 2 (since present in memory, so bring it to the front of the queue. This is called hit)



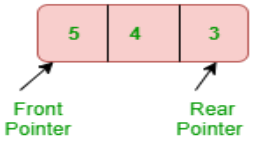
Input : 3 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.)



Input : 4 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.)



Input : 5 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.)



So, we have only 2 hits and 10 page faults using LRU page replacement algorithm.

Efficiency of objects cache

The effectiveness of cache may be statistically determined [8], by calculating a hit

ratio, which depends upon implementation of model of objects cache administration and

means the percent of objects interrogations that the cache administrator is obtaining to.

For example the hit ratio H means the number of accesses finding data within cache from

100 accesses. Miss ratio M – failure percent is calculated as:

$M = 1 – H$

When hit time (Th) is the time of reading in cache and Tm (miss time) is the time

lost in case of failure, then the average access time at cache memory will be calculated

bellow:

$T = Th * H + Tm * M$

The Miss time (Tm) equals the reading time from the slow memory (Ts) addition

the cache reading time. The cache will be efficient if $T<Ts$.

The program execution has been checked using 20 tests applied to the LRU and

LRU-H algorithms for various size of cache. The results of tests enabled to determine the

average values of parameters H, M, Th, Tm and T.

The comparative analysis of average values of parameters H, average access time of

cache (T), the access time of slow memory and the size of cache suggests the

effectiveness of implementation of objects cache (table 1), using the LRU-H algorithm

due to:

• Hit ratio (H) for LRU-H algorithm is higher than for LRU algorithm, meaning that

LRU-H algorithm finds more objects within cache.

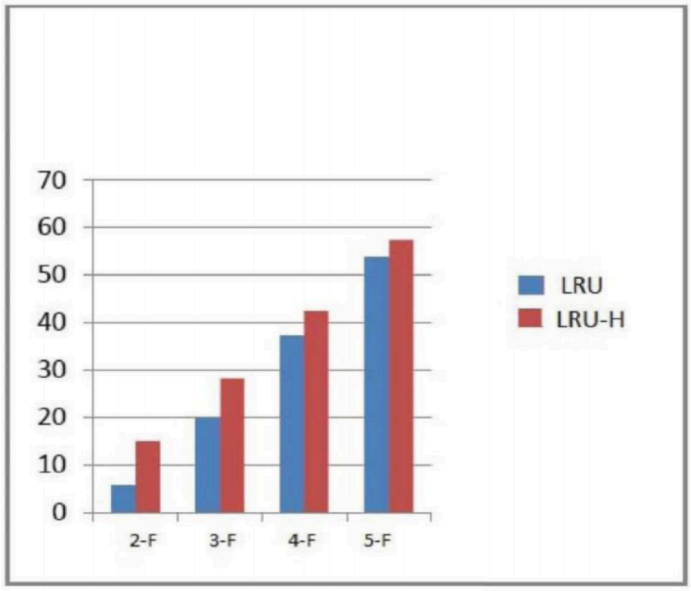
• The access average time at cache memory (T) of tests using LRU-H algorithm is less

than of tests using LRU algorithm.

Table 1. Performing Characteristics of Algorithms LRU and LRU-H

algorithm average
number of
objects found
in cache
H M Th Ts Tm T
(msec)
LRU 8.25 (total 21
references)
39.28% 60.72% 125 655 780 522.716
cache
10
objects
LRU-H 10.15 (total 21
references)
48.33% 51.67% 141 577 718 439.135
390
LRU 15.37 (total 40 references)
38.42% 61.58% 287 1289 1576 1080.76
cache
20
objects
LRU-H 18.42 (total 40 references)
46.05% 53.95% 316 1146 1462 934.267

COMPARISON OF LRU&LRU-H.



BAR I
WORKING OF CLOCK PRO.

As we can infer that when we consider k references we allow the entries which are frequently used or occurring thus helping to reducing the number of page faults in the LRU algorithm. This significantly improves the performance by a 10-13% improved cache references and the 5-10% less page faults .

The improved LRU-H or LRU(m) is used widely in website server where they need to reduce the server request time to provide faster responses to the server requests from the users.

7.REFERENCES

- **Cphash: A cache-partitioned** hash **table**
- LRU**-based column-associative caches**
- **Modified pseudo** LRU **replacement algorithm**
- **Benchmark synthesis using the** LRU **cache hit function**

-END-