# Least Recently used (LRU) cache

November 21, 2021

**Vinay kumar (2020csb1141) ,**
**Yadwinder singh (2020csb1143) ,**
**Vijay dwivedi (2020csb1140)**

**Instructor:**
Dr. Anil Shukla

**Teaching Assistant:**
Ravi Bhatt

**Summary:** The aim of the lab is to simulate an LRU Cache algorithm which gives you the least recently accessed and most recently accessed page in constant time using queue data structure, doubly linked list and hash table.

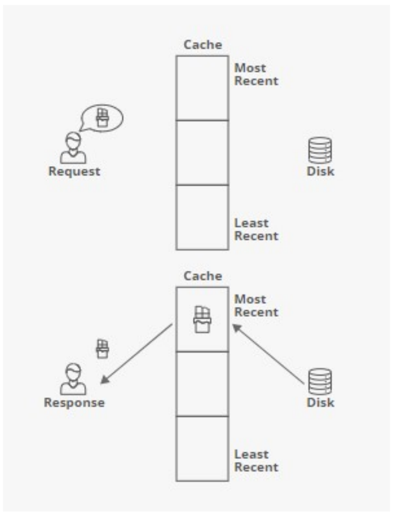
## 1. Introduction

Let Say you're managing a cooking site with lots of cake recipes. As with any website, you want to serve up pages as fast as possible.
When a user requests a recipe, you open the corresponding file on disk, read in the HTML, and send it back over the network. This works, but it's pretty slow, since accessing disk takes a while.
Ideally, if lots of users request the same recipe, you'd like to only read it in from disk once, keeping the page in memory so you can quickly send it out again when it's requested. Bam. You just added a cache.
A cache is just fast storage. Reading data from a cache takes less time than reading it from something else (like a hard disk).

Here's one idea: if the cache has room for, say, n elements, then store the n elements accessed most recently.
To make this concrete, say we have these four recipes on disk:

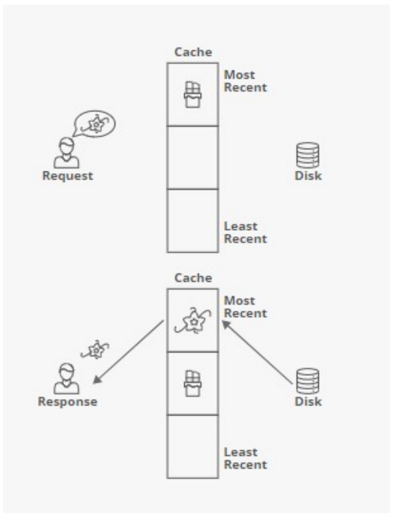
.

Let's say our cache can only store up to three recipes . Let's walk through what the cache might look like over time. First, a user requests the chocolate cake recipe. We'll read it from a disk, and save it to the cache before returning it it the user.



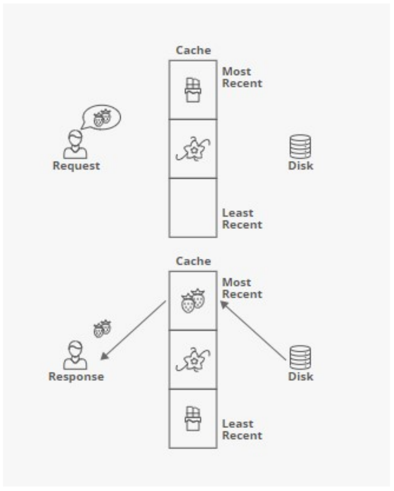
.

Next, someone requests the vanilla cake recipe:



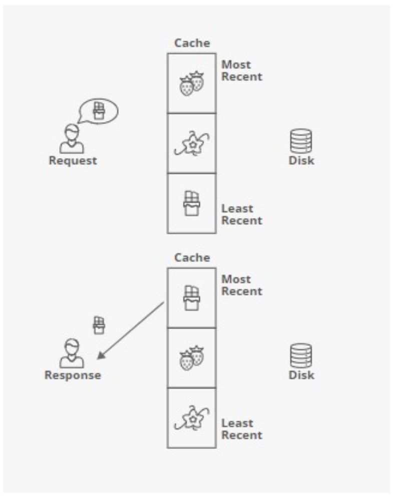
.

Notice that the chocolate cake recipe got bumped down a level in the cache - it's not the most recently used anymore. Next comes a request for the strawberry cake recipe:



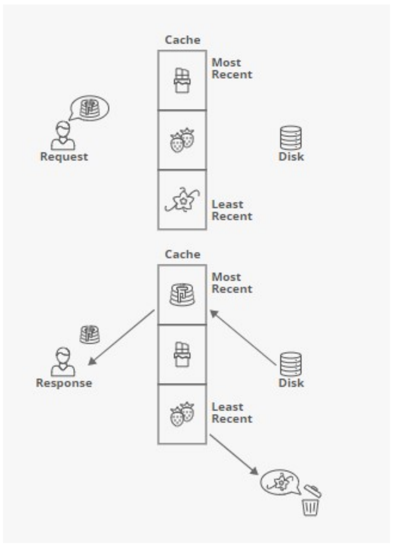
,

And one for chocolate:



.

We had that one in the cache already, so we were able to skip the disk read. We also bumped it back up to the most recently used spot, bumping everything else down a spot. Next comes a request for the pound cake recipe:



.

Since our cache could only hold three recipes, we had to kick something out to make room. We got rid of ("evicted") the vanilla cake recipe, since it had been used least recently of all the recipes in the cache. This is called a "Least-Recently Used (LRU)" eviction strategy.

---

## 2. Equations

Suppose we have n items and map each to one of k slots. We assume the n choices of a slots are independent. A collision is the event that an item is mapped to a slot that already stores an item. A possible resolution of a collision adds the item at the end of a linked list that belongs to a slot, but there are others. We are interested in the following quantities:

**2.1 The expected number of items mapping to same slot;**
Since all slots are the same and none is more preferred than any other, we might as well determine a expected number of items that are mapped to slot 1. Consider the corresponding indicator random variable

$$X_i = \begin{cases} 1 & \text{if item } i \text{ is mapped to slot 1;} \\ 0 & \text{otherwise.} \end{cases}$$

The number of items mapped to slot 1 is therefore X = $X_1 + X_2 + ... + X_n$.
$Theexpectedvalueof X_i is \frac{1}{k}, foreachi. Hence, theexpectednumberofitemsmappedtoslot1is$

$$E(x) = \sum_{i=1}^{n} E(Xi) = \frac{n}{k}, \tag{1a}$$

**2.2 The expected number of empty slots;**
The probability that slot j remains empty after mapping all n items is (1  1 k ) n. Defining

$$X_j = \begin{cases} 1 & \text{if slot } j \text{ remains empty;} \\ 0 & \text{otherwise,} \end{cases}$$

we thus get

$$E(Xj) = (1 - \frac{1}{k})^n, \tag{2a}$$

)n. The number of empty slots is X = $X_1 + X_2 + ... + X_k . numberofemptyslotsis$

$$E(x) = \sum_{j=1}^{n} E(Xj) = k(1 - \frac{1}{k})^n, \tag{3a}$$

For k = n, we have $\lim_{n \to +\infty} (1 - \frac{1}{n})^n = \frac{1}{e} = 0.367$
In this case, we can expect about a third of the slots to remain empty.

**2.3 The expected number of collisions;**
The number of collisions can be determined from the number of empty slots. Writing X for the number of empty slots, as before, we have k  X items hashed without collision and therefore a total of n  k + X collisions. Writing Z for the number of collisions, we thus get

$$E(Z) = n - k + E(X) = n - k + k(1 - \frac{1}{k})^n \tag{5a}$$

For k = n, we get $\lim_{n \to +\infty} n(1 - \frac{1}{n})^n = \frac{n}{e}$
In other words, about a third of the items cause a collision.

## 3. Figures, Tables and Algorithms

### 3.1. Figures

The given graph shows a comparison between LRU, FIFO and OPTIMAL page replacement algorithm on the basis of the hit ratio; hit ratio is defined as the percentage of the objects being found in the cache when requested.

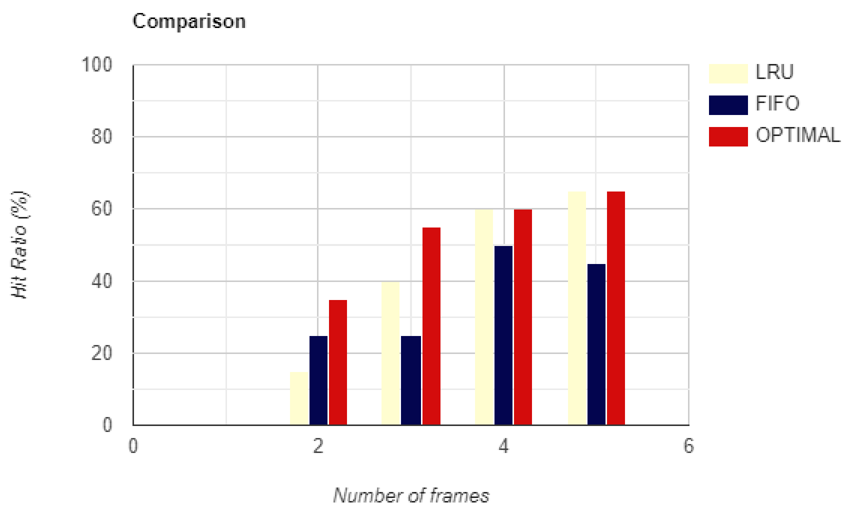


Figure 1: Comparison of LRU, FIFO and OPTIMAL .

### 3.2. Tables

Time complexity is the amount of time taken by an algorithm to run, as a function of the length of the input. Here, the length of input indicates the number of operations to be performed by the algorithm. Below Table compares the time complexities of various operations.

| . | **Time complexity** |
|---|---|
| storing an entry | O(1) |
| searching an entry | O(n) [worst case] |
| removing LRU | O(1) |

## 3.3. Algorithms

### 3.3.1 pseudo code

Algorithm 1 LRU cache algorithm

```
1: Page refreshing (Hash table, String)
2: if already present in Hash table //search for the string in hash table then
3:    Move it to the head of queue // make it the most recently accessed element
4: else
5:    Hash the string
6:    if Queue is full then
7:       Dequeue() // remove the least recently accessed element
8:    else
9:       Enqueue()
10:   end if
11: end if
```

### 3.3.2 IMPLEMENTATION

An LRU cache is built by combining two data structures: Queue (using doubly linked list) and a hash map(hashing by chaining). The usage of doubly linked list makes the operations like deleting a node, adding a new node very easy and efficient. We set up our linked list with the most-recently used item at the head of the list and the least- recently used item at the tail. This lets us access the LRU element in O(1) time by looking at the tail of the list. In general, finding an item in a linked list is O(n). But the whole point of a cache is to get quick lookups. We add in a hash map that maps items to linked list nodes. It reduces the time dependency of search from the size of cache to the size of slot in hash table.

We made following functions to implement algorithm.

**int value()**
-> The function takes a string as input and return the sum of ASCII values of last five characters of the string. The function is a part of the hash function we are using for hashing. It always runs a loop of five steps, hence it is a constant time operation i.e O(1).

**void hash-insert()**
-> When the program takes a string as input, it inserts the string in the hash table. The hash function provides a particular slot number, the function adds a new node at the head of that slot. Basically, we are using hashing by chaining. The time complexity of the operation is O(1).

**int hash-search()**
-> The function tells you whether a particular string is already present in our database or not. It returns 1 if it is already present and 0 it not. The hash function tells the slot number where it can be possibly found. It traverses the list of the slot and search for the string. The time of search depends on the number of elements in the list of that slot. In worst case it is O(n).

**void hash-delete()**
-> Function removes the node from the hash table. It goes to the slot where it can be possibly found and if present removes it from the list of that slot. The time complexity of the operation is O(1).

**void enqueue()**
-> The function adds a new item in the queue. It allocates the memory for the node and add at the rear of the doubly linked list and makes the rear equal to the new node added. The time complexity of the operation is O(1).

**char *dequeue()**
-> When you need to add a new node in queue but the queue is already full, Dequeue is called. It removes the item at the front of queue (which is the least recently used item) and returns the same. The time complexity

of the operation is O(1).

**void referencePage()**
-> This is the function, which uses all the functions simultaneously and implements the algorithm. It takes the string and use hash-search function to search for the string. If the string is already present, it moves the string node to the tail of the queue and makes it least recently used item and if it is not present then it adds the item into hash table and at the rear of the Queue. Its time complexity depends only on the hash-search function. So we can say that, it also takes O(n) time in worst case scenario.
.

---

### 3.3.3 EXPLANATION

**length cache = 4**

**size hashtable = 5**

.

step1:
Insert (https://ddabc)
**Queue**
https://ddabc
d=100,d=100 ,a=97 , b=98 , c=99
100+100+97+98+99=494
**Hash table**
0
1
2
3
4 ->https://ddabc

step2:
Insert (https://!Cdef)
**Queue**
https://!Cdef https://ddabc
!=33,C=67,d=100 , e=101 , f=102
33+67+100+101+102=403
**Hash table**
0
1
2
3 ->https://!Cdef
4 ->https://ddabc

.

step3:
Insert (https://,8ghi)
**Queue**
https://,8ghi https://!Cdef https://ddabc
,= 44 , 8=56 , g=103 , h=104 , i=105
44+56+103+104+105=412
**Hash table**
0
1
2 ->https://,8ghi
3 ->https://!Cdef
4 ->https://ddabc

step4:
Insert (https://Adgf)
**Queue**
https://Adgf https://,8ghi https://!Cdef https://ddabc
=35 , A=65 , d=100 , g=103 , f=102
35+65+100+103+102=405
**Hash table**
0 ->https://Adgf
1
2 ->https://,8ghi
3 ->https://!Cdef
4 ->https://ddabc

.

step5:
Insert (https://!Cklm)
Deque – https://ddabc.
**Queue**
https://!Cklm https://Adgf https://,8ghi
https://!Cdef
!=33 , C=67 , k=107 , l=108 , m=109
33+67+107+108+109=424
**Hash table**
0 ->https://Adgf
1
2 ->https://,8ghi
3 ->https://!Cdef
4 ->https://!Cklm

step6:
Insert (ghi)
Already available in the hashtable
Deque - https://,8ghi
**Queue**
https://,8ghi https://!Cklm https://Adgf
https://!Cdef
,= 44 , 8=56 , g=103 , h=104 , i=105
44+56+103+104+105=412
No change in hashtable
**Hash table**
0 -> https://Adgf
1
2 -> https://,8ghi
3 -> https://!Cdef
4 -> https://!Cklm

.

.

## 4. Conclusions

In this report, we discussed the implementation of **LRU cache eviction policy**. The algorithm shows that the least recently accessed element will always be at the rear of the node, so to access you need to just access its data, hence concluding that the algorithm gives the least recently accessed item in **constant time**. Mathematically we got an expected memory distribution of Hash Table. Also on the behalf of the graph results, it's concluded that the LRU algorithm is the best as compare to the all other practical algorithm.

## 5. Bibliography and citations

Following are the sources that were referred to make this report and program credible and refined:

- Lectures notes [3] gave us a basic understanding of implementation of Queue data structure and hash table that we used to implement LRU cache eviction policy.
- Discrete Mathematics for Computer Science by Herbert Edelsbrunner and Brittany Fasy [1] helped us understanding the hashing by chaining more deeply and we were able to give the expected memory distribution in hash table.
- We took the data of comparison of different page replacement policies from the article by Ambreen Malik Imran Anwar Muhammad Waqar, Anas Bilal [2] which gives a clear proof that LRU cache eviction policy is fast as compared to other policies.

## References

[1] Herbert Edelsbrunner and Brittany Fasy. Discrete mathematics for computer science. 2009.

[2] Ambreen Malik Imran Anwar Muhammad Waqar, Anas Bilal. Comparative analysis of replacement algorithms techniques regarding to technical aspects, 2016.

[3] Dr. Anil Shukla. *CS201 Class Notes.*

.