**Design Document for Cache project**

**What is a cache?**

A cache is a place to store something temporarily in a computing environment. It is used to shorten data access times, reduce latency and improve input/output. In [computing](https://en.wikipedia.org/wiki/Computing), a **cache** is a component that stores data so future requests for that data can be served faster; the data stored in a cache might be the result of an earlier computation, or the duplicate of data stored elsewhere. A cache hit occurs when the requested data can be found in a cache, while a cache miss occurs when it cannot. Cache hits are served by reading data from the cache, which is faster than recomputing a result or reading from a slower data store; thus, the more requests can be served from the cache, the faster the system performs

**API Of Cache**

JCache

JCache is a Java API for caching. It provides a set of common interfaces and classes that can be used to temporarily store Java objects in memory. It is a JCP standard represented by JSR 107. The JCache API is implemented by different but the manner in which client code accesses it is via a standard set of interfaces and classes that it exposes.

**Cache Interface**

The Cache interface exposes several features via its API, but some of the basic ones are listed in the following table:

|  |  |
| --- | --- |
| METHOD | DESCRIPTION |
| put , putAll | Adds entries (key-value pairs) to a cache |
| containsKey | Checks if an entry with a key exists in the cache |
| get , getAll | Gets value(s) corresponding to key(s) |
| replace | Substitutes an existing value for a key with another value |
| remove , removeAll | Removes one or all entries from a cache |

**CacheManager**

The javax.cache.CacheManager interface helps deal with cache objects and executes operations like cache creation, destruction, and introspection (fetching relevant details about it). Let’s look at some of the common operations:

|  |  |
| --- | --- |
| METHOD | DESCRIPTION |
| createCache | Create a cache |
| destroyCache | Destroy a cache |
| getCache (and its overloaded forms) | Search for a (possibly) existing cache |
| getProperties, getURI, isClosed, getClassLoader, etc. | Provide information about the cache |

**Cache Eviction Algorithm**

A cache eviction algorithm is a way of deciding which element to evict when the cache is full. The following are the different cache eviction algorithm

1. Bélády's Algorithm
2. Least Recently Used (LRU)
3. Most Recently Used (MRU)
4. Pseudo- LRU (PLRU)
5. Random Replacement (RR)
6. Segmented LRU (SLRU)
7. 2-way set associative
8. Direct-mapped cache
9. Least-Frequently Used (LFU)
10. Low Inter Reference Recency Set (LIRS)
11. Adaptive Replacement Cache (ARC)
12. Clock With Adaptive Replacement (CAR)

### First In First Out (FIFO)

* Elements are evicted in the same order as they come in. When a put call is made for a new element (and assuming that the max limit is reached for the memory store) the element that was placed first (First-In) in the store is the candidate for eviction (First-Out).
* This algorithm is used if the use of an element makes it less likely to be used in the future. An example here would be an authentication cache.

### Least Frequently Used (LFU)

* For each get call on the element the number of hits is updated. When a put call is made for a new element (and assuming that the max limit is reached) the element with least number of hits, the Least Frequently Used element, is evicted.
* If cache element use follows a pareto distribution, this algorithm may give better results than LRU

### Least Recently Used (LRU)

* This is the default and is a variation on Least Frequently Used.
* The oldest element is the Less Recently Used (LRU) element. The last used timestamp is updated when an element is put into the cache or an element is retrieved from the cache with a get call

### Data structure used to implement FIFO Cach

1. The FIFO cache algorithm is the most primitive type of algorithm. In this algorithm the element is evicted based on the manner in which it is inserted.
2. Queue is used for this purpose.

**Data Structures used to implement LRU Cache**

**We use two Data Structures to implement LRU Cache**

1. A 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 rear end.
2. A Hash with page number as key and address of the corresponding queue node as value.

### Data structure used to implement LFU Cache

It requires three data structures.

1. One is a *hash table* which is used to cache the key/values so that given a key we can retrieve the cache entry at O(1).
2. Second one is a double linkedlist for each frequencyof access. The max frequency is capped at the cache size to avoid creating more and more frequency list entries. If we have a cache of max size 4 then we will end up with 4 different frequencies. Each frequency will have a double linked list to keep track of the cache entries belonging to that particular frequency.
3. The third data structure would be to somehow link these frequencies lists. It can be either an arrayor another linked list so that on accessing a cache entry it can be easily promoted to the next frequency list in time O(1). In our article it is based on array as traversing would be faster than linked list.