Caching library

1. Introduction

In this project, I will design a caching library from scratch using python. In computing, a cache is a hardware or software component that stores data so that future requests for that data can be served faster; the data stored in a cache might be the result of an earlier computation or a copy of data stored elsewhere[1].

2. Design

A cache design should consider so many factors like size, speed etc. There is an inherent trade-off between size and speed (given that a larger resource implies greater physical distances) but also a tradeoff between expensive, premium technologies (such as SRAM) vs cheaper, easily mass-produced commodities [1].

2.1 Cache

I set the data type of cache as defaultdict(set). Because, the speed of defaultdict much faster than normal dict in python, especially when the dataset is very huge. As we know, the default value for defaultdict(set) is {}. We can simplely check its key's existence by calling "not cache[key]" which will return True if the key doesn't exist. Besides, a defaultdict will never raise a KeyError. Any key that does not exist gets the value returned by the default factory.

2.2 FIFO Queue

In this project, I use a FIFO queue to manage the sequence of the key, value pair. FIFO stands for first-in, first-out. It's an algorithm which used by the stack memory. When the stack memory is full, some data of the memory need to be cleared to stored new data, as figure 1 shows. FIFO – an acronym for first in, first out – in computing and in systems theory, is a method for organizing the manipulation of a data structure – often, specifically a data buffer – where the oldest (first) entry, or 'head' of the queue, is processed first [2].

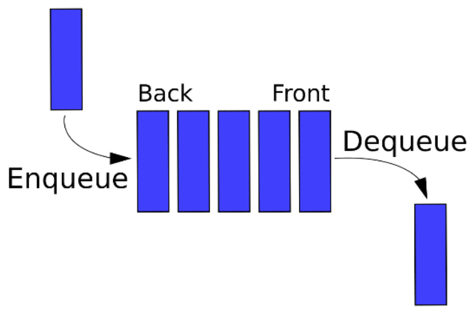

Figure 1. Structure of FIFO

This speed would benefit from this type of data structure, is because when the cache is full, the first element in the queue, which represents the key of the oldest value will be retrieved without any delay. Otherwise, the program needs time to compare each key-value pair's datetime to find the oldest pair, which is time-consuming.

2.3 Cache Size

The size of the cache would affect the speed performance of the caching significantly. If the size is very huge, then the speed of the cache will be slowed down greatly. However, if the size is very limited, then the speed will be negatively affected as well. Because the cache will be fully occupied easily, when new key-value comes in, the cache has to remove the oldest record first which will take additional time to implement. In this project, I set the value of cache size as 10e6, which could hit a balance between size and speed.

3. Implementation

To implement this project, I defined the following functions – insert, update, delete and remove. These functions are aimed to manipulate key-value pair data in cache.

3.1 Insert

This insert function takes a key-value pair as the input. As the figure 2 below shows, it will check the status (full or not) of the cache first before insertion. If the cache is not full, it will insert the key and value with the current date-time into the cache. Then, it will append the key to the FIFO queue. In case the cache is full, it will call the remove_redundance() function to delete the oldest record before insertion.

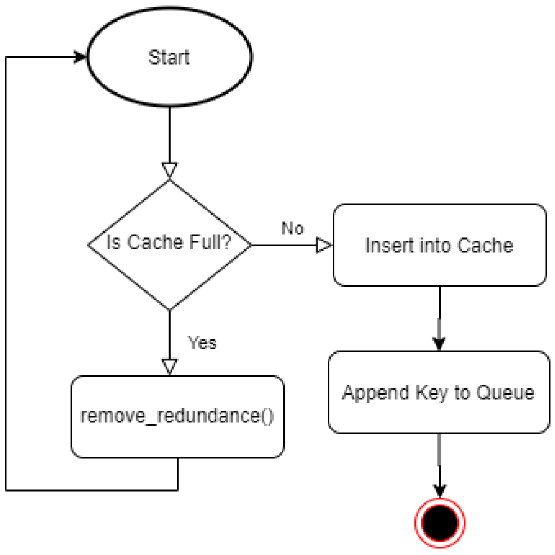

Figure 2. Insert Workflow

3.2 Update

Like the insert(), update function also take key-value pair as its input. First, it will check the existence of the key. If the key doesn't exist, the function will return the "Key not found" message. Otherwise, the new value will overwrite the original one, and at the same time, the datetime will be updated to the current date-time as well. After that, the key will be removed from the FIFO queue, before appending to the tail of the queue.

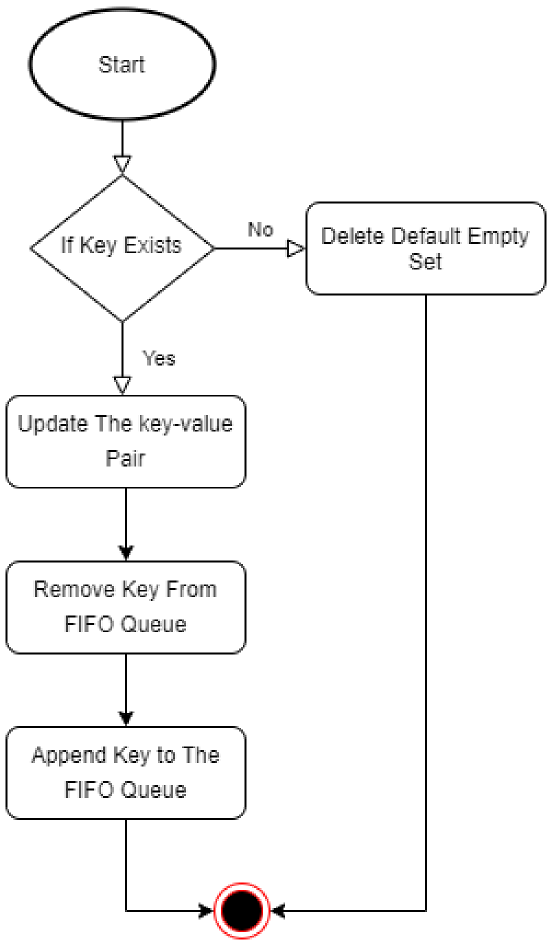
Start
If Key Exists
No
Delete Default Empty Set
Yes
Update The key-value Pair
Remove Key From FIFO Queue
Append Key to The FIFO Queue

Figure 3. Update Workflow

3.3 Delete

Figure 4 below depicted the workflow of the deletion process. The delete function takes only key as its parameter. It tries to remove the key from the FIFO queue, before calling cache.pop(key) to remove the record from the cache by key. However, two potential exceptions - "KeyError" and "ValueError" will be raised, if either "key not found" or "value not found".

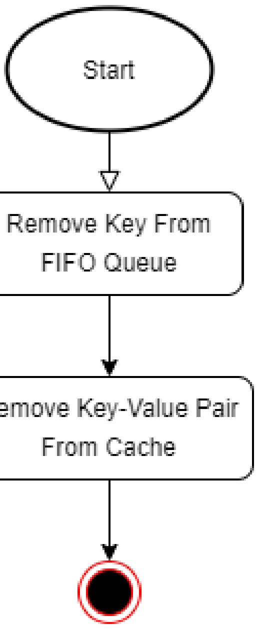

Figure 4 Delete Workflow

3.4 Remove

The remove_redundance is designed to remove the oldest record from the cache, in case the case is fully occupied. Benefited from FIFO design, the key of the oldest record is always stored as the first element of the queue. So, to retrieve the targeted key, let's call function queue.pop(0). Finally, we can remove the key-value pair from the cache by calling cache.pop(k).

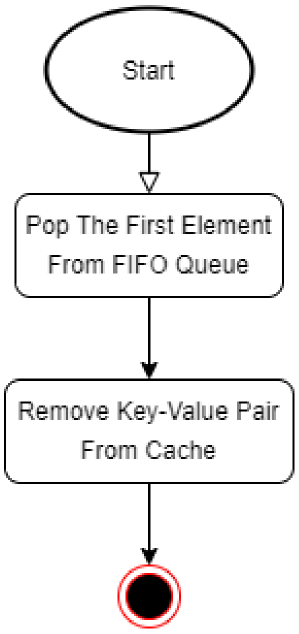
Start
Pop The First Element From FIFO Queue
Remove Key-Value Pair From Cache

Figure 5 Remove Workflow

4. References

[1] En.wikipedia.org. 2020. Cache (Computing). [online] Available at: <https://en.wikipedia.org/wiki/Cache_(computing)> [Accessed 17 June 2020].

[2] En.wikipedia.org. 2020. FIFO (Computing And Electronics). [online] Available at: <https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics)> [Accessed 17 June 2020].