CET - Data Structures and Algorithms with Java - Practicum

Topics Covered: Classes, OOP, Singly Linked List

Learning Objectives:

- Familiarize with the workings of a singly linked list data structure.
- Understand and implement the cache functionality of a computer using a singly linked list from scratch.

Deliverables:

- Submit a single zip file called `CET_P0_<Your_Name>.zip` (e.g. `CET_P0_John_Doe.zip`) containing your Java project, include your driver class.
- Note that a non-working submission will result in a zero

Background

Modern operating systems uses several CPU caches to reduce the average cost (time or energy) to access data from main memory. A typical cache is a small, very fast memory located close to the CPU. Its sole purpose is to store copies of frequently used data from main memory. Generally, there is a hierarchy of multiple cache levels (L1, L2 & L3). The size of these caches are also limited.

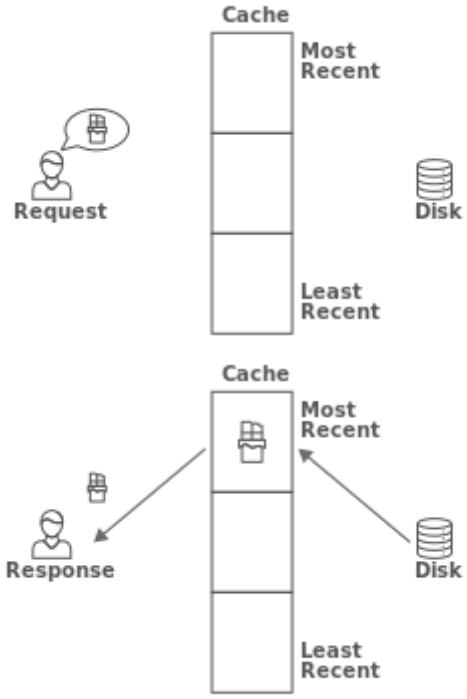
When the CPU is looking for data, it would first search in the L1 cache before moving to the lower levels and finally to the main memory. The operating system manages the cache memory so that it can store new files and remove unwanted or least used files. The algorithm used is known as the eviction policy. There are 2 types of eviction policies:

- Least Recently Used (LRU)
- Most Recently Used (MRU)

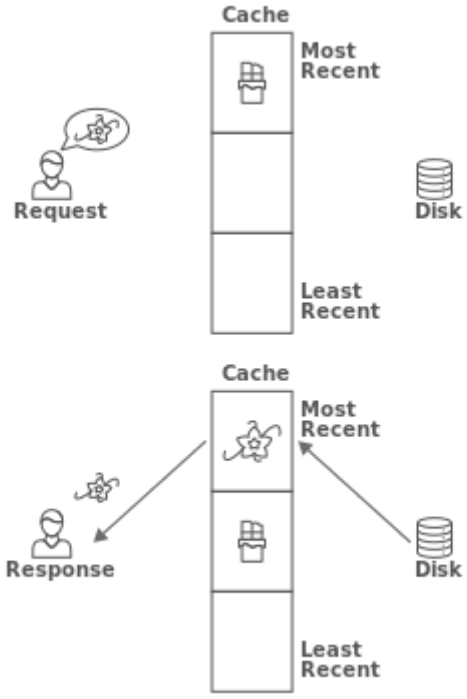
Eviction Policies

The following is an explanation of the LRU eviction policy where it removes the *least recently* used data from the end of the cache. Let's use some pictures to help us understand this. Let's say that we have 4 recipes on disk (figure below) and a cache that can only store 3 recipes at any point in time.

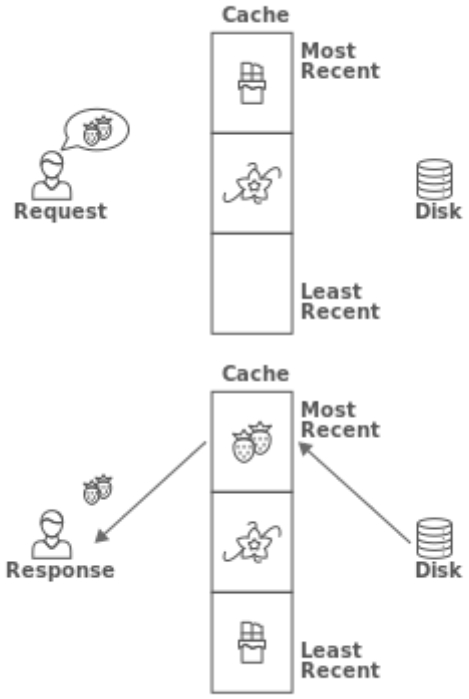
Suppose a user requests the chocolate cake recipe. It is read from disk and saved to the cache before returning it to the user.



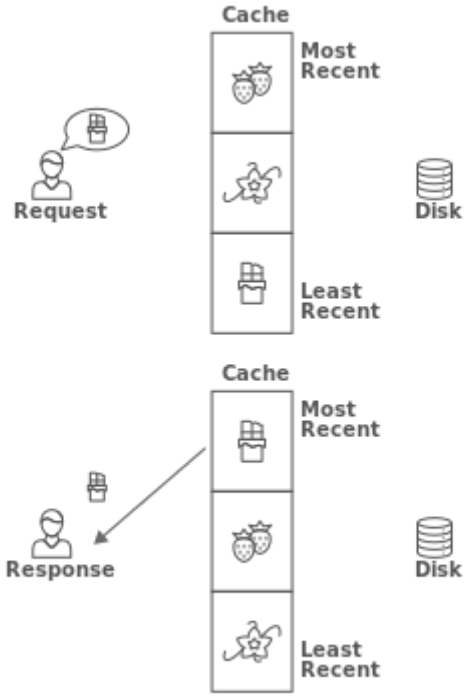
Next, another user requests the vanilla cake recipe.



Notice how the chocolate cake recipe has now moved down a slot in the cache, it is no longer the most recently used recipe anymore. Next a request for the strawberry cake is received:

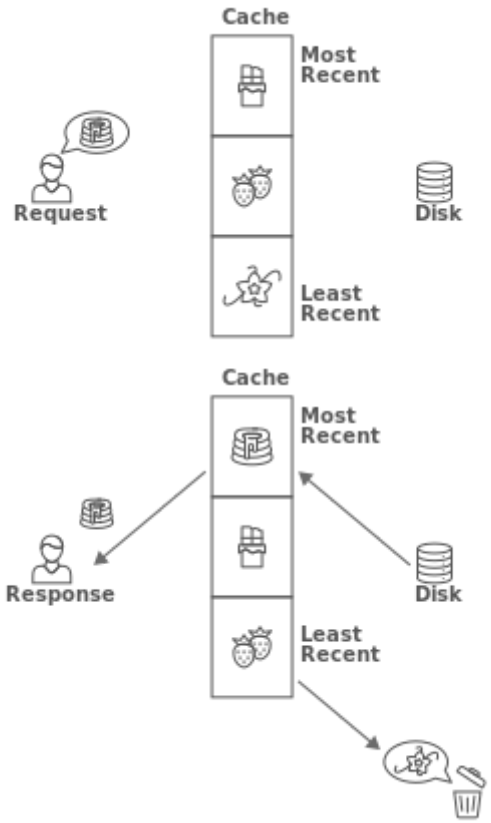


and another for the chocolate cake recipe.



Since the chocolate cake recipe is already in the cache, the disk read is skipped. In addition, the chocolate cake recipe is moved up to the most recently used spot, moving everything else down a slot.

Next, comes a request for the pound cake recipe:



Since the cache can only store 3 recipes, a recipe has to be removed from the cache to make space for the pound cake recipe. Because the cache is using the **LRU eviction policy**, it will **remove** the vanilla cake recipe (aka remove from the **tail**). Whereas for **MRU eviction policy**, the most recently added recipe will be removed (aka remove from the head). Everything other functionality is the same.

Tasks

Provided is a **template of classes** from which you are to complete based on the following requirements:

- You are to read the comments in the template code carefully before starting. **Only** fill in the areas with the comment: `// YOUR CODE HERE`.
- All class attributes are private thus use the appropriate **accessor functions**. Not all accessor functions are required.
- Create your own driver class with a `main` function and submit this file with your Java project.
- The data is stored as using the `ContentItem` class consisting of the following attributes:
  - content id: Integer
  - size: Integer
  - header: string
  - content: string

  not primitive take note
- There can be **more than** 2 levels of caches with a **default of 3 cache levels** when the number of cache level is not defined.

  L1, L2, L3 caching

- The directory of the caches is to be implemented as a hash table. This means that to determine which cache level the data belongs to, we use the value in the content header for the hash function to produce an index for the appropriate cache level. Eg: If the content header contains "Type 1", it sums up to 499 and if there are 3 cache levels, 499 % 3 = 1. Thus that `ContentItem` object would be inserted into the L2 cache.
- Insertion, eviction and retrieval of `ContentItem` objects within a cache is to be implemented as a **singly linked list**.
- Each level's cache has a **maximum size of 200**. This size refers to the sum of all the `size` attribute of each `ContentItem` object in the cache. It is an arbitrary number that is **not related** to the actual size in bytes of the content.
- The eviction policy is represented by a string, either '**lru**' or '**mru**'.
- No 2 `ContentItem` objects with the same id can exist within the same cache.
- When the content in the `ContentItem` object is updated, **only** the content is updated and nothing else. You do not need to worry about managing the new content sizes. The replaced content will always have the same size as the content before.
- Appropriate error messages should **only** be shown by the driver class. In other words, there are no print statements other than in the driver class.
- Include any other methods that you many require.
- Other than the given Java libraries, **no other in-built or 3rd party Java libraries are allowed**.

do algo on eviction policy first! can ask for advice early

first content item got 100, second content item got 150 total got 250, but maximum only 200 allowed, so need to evict ppl out

Rubrics

For the maximum allocation of marks, refer to the table below.

Description	Marks (%)
Proper implementation of the `ContentItem` & `Node` classes	10
Proper implementation of the `CacheList` & `LinkedList` class	25
Proper implementation of the `Cache` class	10
Successful integration of the program which includes but not limited to: - passing instructor's test case - no bugs - does not crash	55

Once you have completed, zip your files and rename it in the following format `CET_P0_<Your_Name>.zip` e.g. `CET_P0_John_Doe.zip`.

Reference

1. LRU Cache, Interview Cake, https://www.interviewcake.com/concept/java/lru-cache