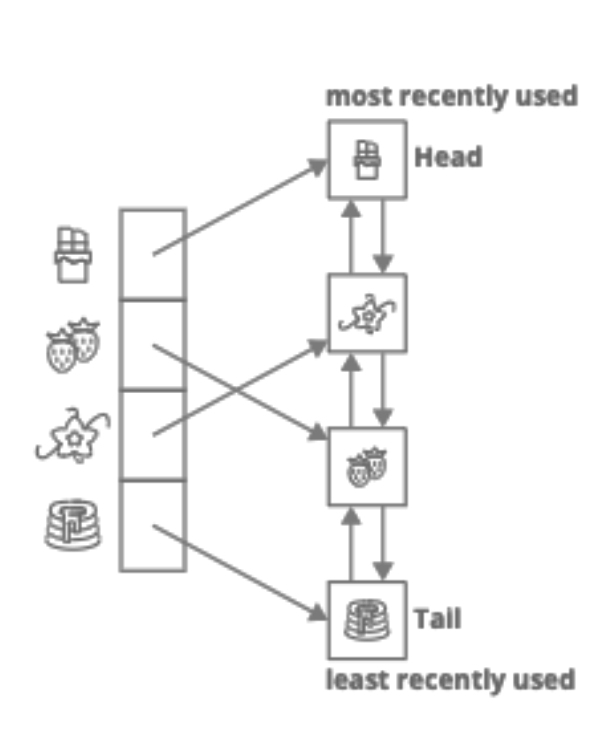


LRU Cache

Data Structure

Quick reference

A **Least Recently Used (LRU) Cache** organizes items in order of use, allowing you to quickly identify which item hasn't been used for the longest amount of time.

Picture a clothes rack, where clothes are always hung up on one side. To find the least-recently used item, look at the item on the other end of the rack.

Under the hood, an LRU cache is often implemented by pairing a doubly linked list with a hash map.

Costs

	Worst Case
space	$O(n)$
get least recently used item	$O(1)$
access item	$O(1)$

Strengths:

- **Super fast accesses**. LRU caches store items in order from most-recently used to least-recently used. That means both can be accessed in $O(1)$ time.
- **Super fast updates**. Each time an item is accessed, updating the cache takes $O(1)$ time.

Weaknesses

- **Space heavy**. An LRU cache tracking n items requires a linked list of length n, *and* a hash map holding n items. That's $O(n)$ space, but it's still two data structures (as opposed to one).

Why Use A Cache?

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 the cache catch: caches are small. You can't fit *everything* in a cache, so you're still going to have to use larger, slower storage from time to time.

If you can't fit everything in the cache, how do you decide what the cache *should* store?

LRU Eviction

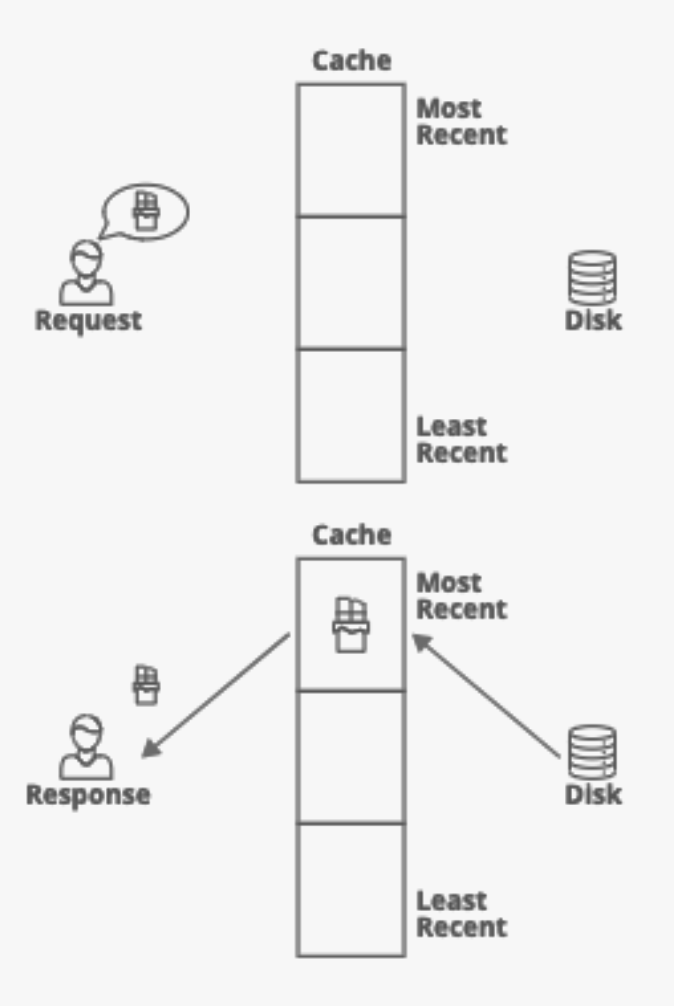
Here's one idea: if the cache has room for, say, n elements, then store the n elements accessed most recently.

To make this concrete, say we have these four recipes on disc:

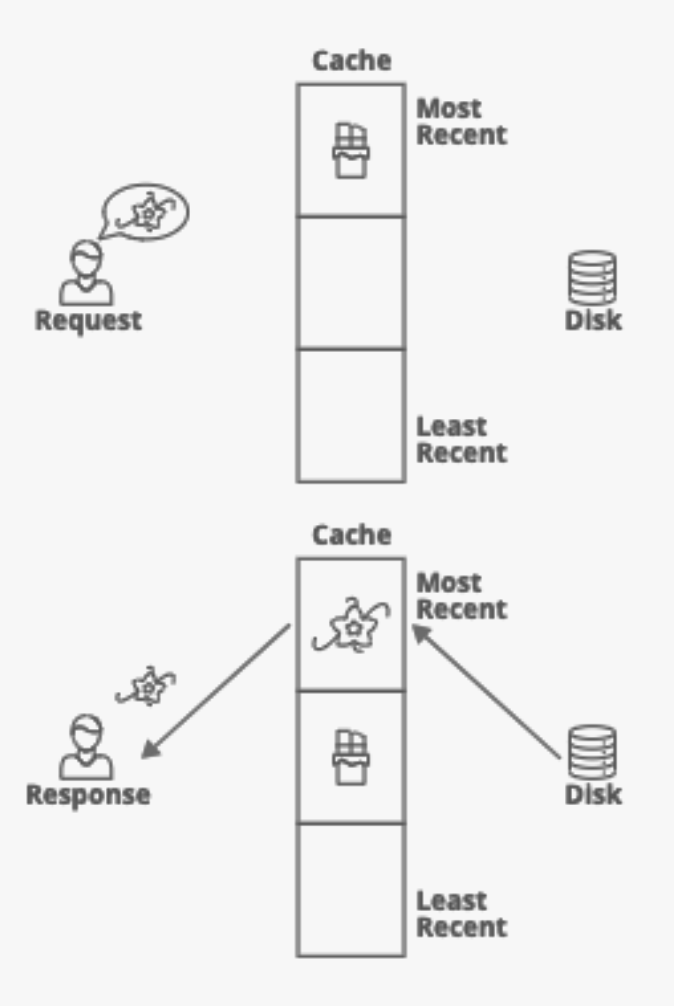
Let's say our cache can only store up to three recipes (that's comically small, but it'll make this example easier to understand).

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 disc, and save it to the cache before returning it it the user.

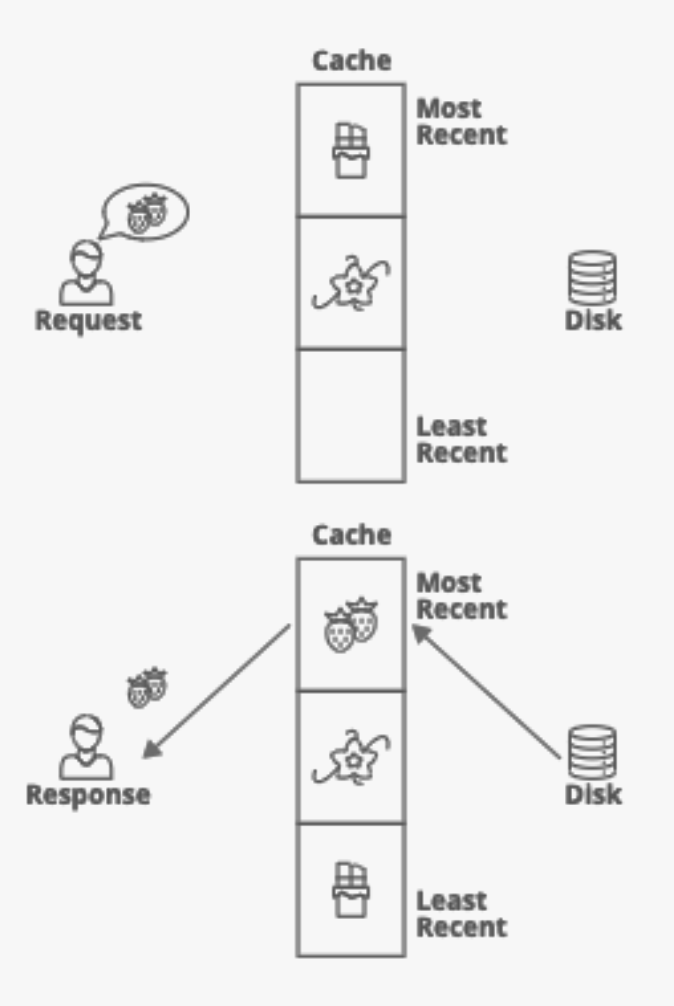


Next, someone requests the vanilla cake recipe:

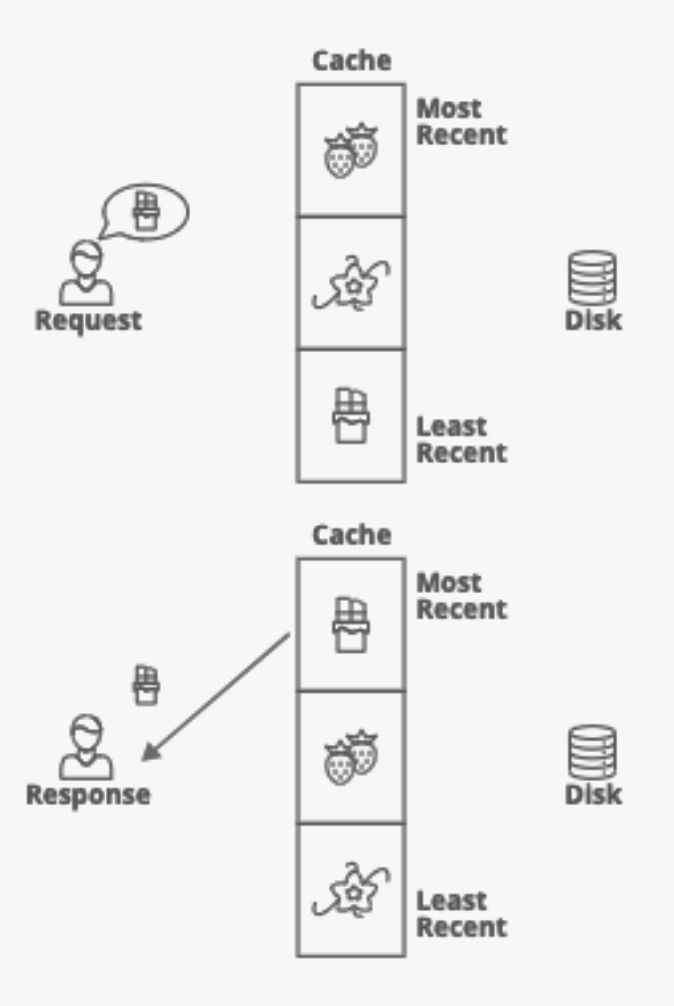


Notice that the chocolate cake recipe got bumped down a level in the cache - it's not the most recently used anymore.

Next comes a request for the strawberry cake recipe:

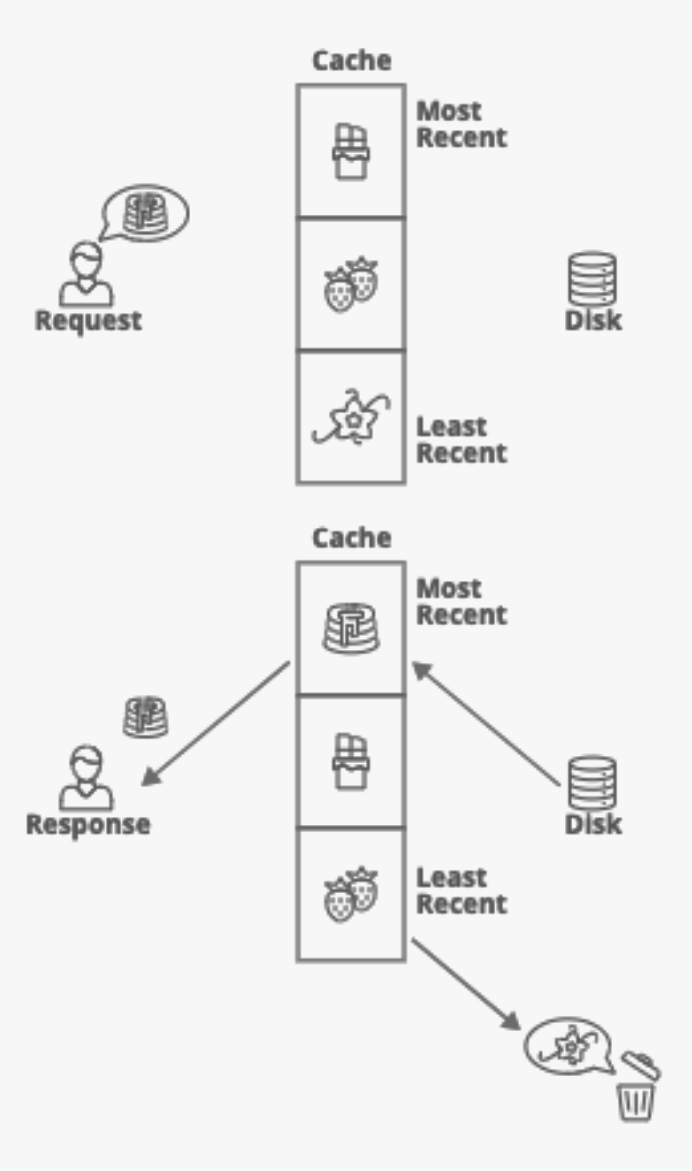


And one for chocolate:



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.**

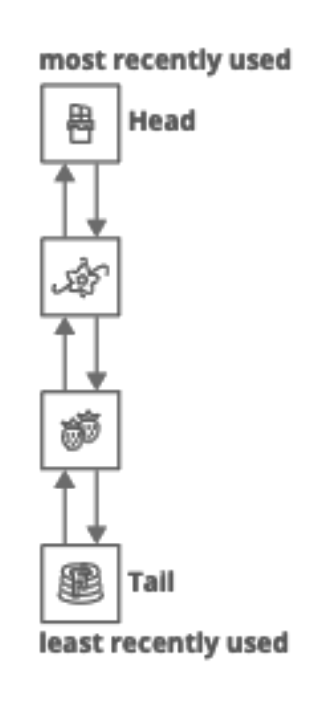
> There are lots of strategies that we could have used to choose which recipe to get rid of. We're focusing on LRU since it's a common one that comes up in coding interviews.

An **LRU cache** is an efficient cache data structure that can be used to figure out what we should evict when the cache is full. The goal is to always have the least-recently used item accessible in $O(1)$ time.

LRU Cache Implementation

An LRU cache is built by combining two data structures: a doubly linked list and a hash map.

We'll 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:

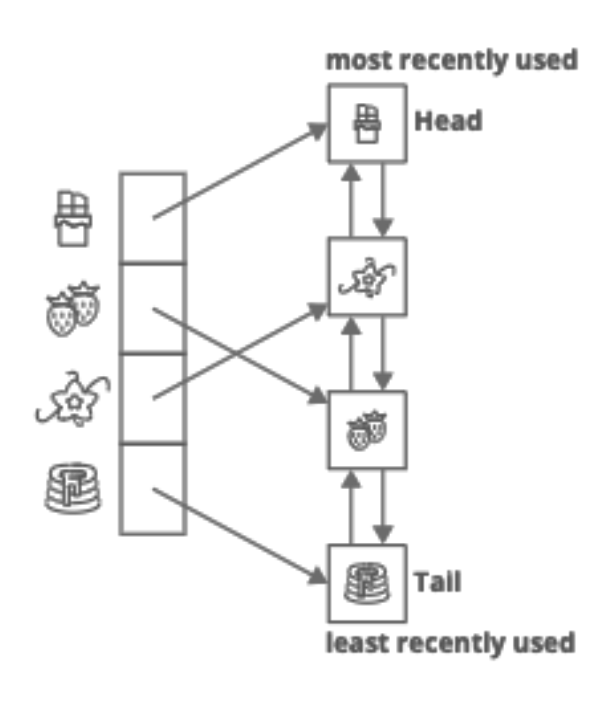


This lets us access the LRU element in $O(1)$ time by looking at the tail of the list.

What about accessing a specific item in the cache (for example, the chocolate cake recipe)?

In general, finding an item in a linked list is $O(n)$ time, since we need to walk the whole list. But the whole point of a cache is to get quick lookups. How could we speed that up?

We'll add in a hash map that maps items to linked list nodes:



That lets us find an element in our cache's linked list in $O(1)$ time, instead of $O(n)$.

Accessing and Evicting

Putting things together, here are the steps we'd run through each time an item was accessed:

- Look up the item in our hash map.
- If the item is in the hash table, then it's already in our cache—this is called a "**cache hit**"
  1. Use the hash table to quickly find the corresponding linked list node.
  2. Move the item's linked list node to the head of the linked list, since it's now the most recently used (so it shouldn't get evicted any time soon).
- If the item isn't in the hash table, we have a **cache miss**. We need to **load** the item into the cache:
  1. Is our cache full? If so, we need to **evict something** to make room:
     - Grab the least-recently used cache item—it'll be at the tail of the linked list.
     - Evict that item from the cache by removing it from the linked list and the hash map.
  2. Create a new linked list node for the item. Insert it at the head of the linked list.
  3. Add the item to our hash map, storing the newly-created linked list node as the value.

> Keeping all the pointers straight as you move around linked list nodes is tricky! Try implementing it yourself! See if you can see why it's important that our linked list is doubly-linked :)

All of those steps are $O(1)$, so put together **it takes $O(1)$ time to update our cache each time an element is accessed**. Pretty cool!

Interview coming up?

Get the free 7-day email crash course. You'll learn *how to think algorithmically*, so you can break down tricky coding interview questions.

No prior computer science training necessary—we'll get you up to speed quickly, skipping all the overly academic stuff.

No spam. One-click unsubscribe whenever.

What's next?

If you're ready to start applying these concepts to some problems, check out our mock coding interview questions.

They mimic a real interview by offering hints when you're stuck or you're missing an optimization.

Try some questions now