MAPS and HASHING:

Sets and Maps: Maps are like sets which doesn’t have the order and doesn’t allow for repeated elements. Sets are kind like a bag, you can reach out in and get whatever you want.

Map is set based data structure map = <key, value>. a group of keys is a set

Everything until now is in linear time

Hashing: Hashing allows to work in constant time.

Value --------------------Hash Function ----------------------------🡪 Hash Value

The purpose of the Hash function is to transform some values in to one that can be stored and retrieved easily. You give it some value, it converts the value based on the formula, and spits out the coded version of the value that often an index of an array.

For example: Let say you are running some big event. Maybe it’s for people who likes certain types of music. Everybody who wants to come in the even needs to buy a ticket in advance. All of those tickets are randomly numbered, and those numbers are embedded on barcodes on physical tickets that people hand you when they first arrive. You need to be able to look at those ticket number fast when people check in at the event. Storing the ticket number in one of those incredible hash functions sounds pretty great.

8675309 -----------------------------------------------🡪 \_\_\_\_\_\_\_\_\_\_\_\_\_

5555555 -----------------------------------------------🡪 \_\_\_\_\_\_\_\_\_\_\_\_\_

0123456 -----------------------------------------------🡪 \_\_\_\_\_\_\_\_\_\_\_\_\_

You have all of your numbers. Now you just need to come up with a hash function to convert them to hash values so they can be stored easily.

One common pattern in hash functions is to take last few digits of a big number, divide it by some consistent number, and using the remainder from that division to find a place to store that number in a array.

Let say the number is 0123456, so take 56 /10 and make remainder (6 here) as an index

Collision: As it turns out there is a flaw in our elegant system. There are times when the hash function will spit out the same values for two different inputs. This situation is termed as collision.

12316 --🡪 remainder 6

12356 --🡪 remainder 6

There are two main ways to fix the collision.

The first is to change the values in your Hash function, or to change the Hash function completely, so you have more than enough slots to store all of your potential values. You can also keep your Hash function but change the structure of your array. Instead of storing one Hash value in the array you can store the list that contains all the Hash values hashed at that spot. These lists are generally called buckets in this context Rather than storing one value in each slot, you can store multiple values or a collection in each bucket, but are either of these approaches helpful?

Hash Maps: Maps have the <key, Value>. You can use the keys as inputs to a hash function, then store the key value pair in bucket of the Hash value produced by the function. Again, since you know the keys in a map are unique since they belong to a set you could use the hash function to give them each their unique buckets. You can also design a Hash function to allow for collisions.

STRING KEYS:

String keys can be used. You just need to come up with hash function that converts letters in to numbers. Individual letter can be easily converted in to Ascii values in many languages already have functions built in that does that. We can combine the Ascii values with a formula to get a unique hash for each letter.

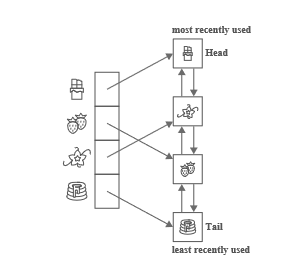
The formula will something looks like this:

S [0] \*31^(n-1) + S [1] \*31^(n-2) + S [2] \*31^(n-3) + … S [n-1]

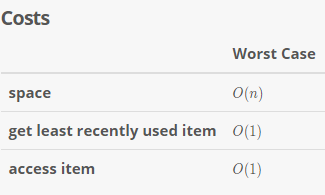
What is Caching?

Caching can be defined as the process of storing data into a temporary data storage to avoid recomputation or to avoid reading the data from a relatively slower part of memory again and again. Thus, caching serves as a fast "look-up" storage allowing programs to execute faster.

Problem 1:



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.



**Strengths:**

Superfast accesses. LRU caches store items in order from most-recently used to least-recently used. That means both can be accessed in O (1) O (1) time.

Superfast updates. Each time an item is accessed, updating the cache takes O (1) O (1) time.

**Weaknesses:**

Space heavy. An LRU cache tracking nn items requires a linked list of length nn, and a hash map holding nn items. That's O(n)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, nn elements, then store the nn elements accessed most recently.

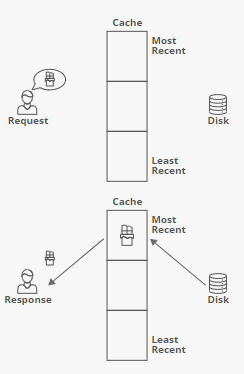
To make this concrete, say we have these four recipes on disk:



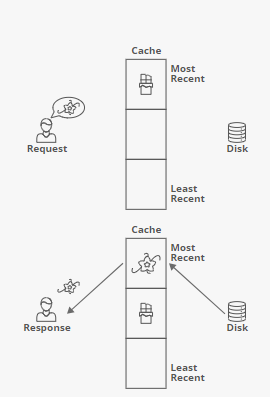
Let's say our cache can only store up to three recipes (that's comically small, but it'll make this example easier to understand).

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 disk and save it to the cache before returning it the user.

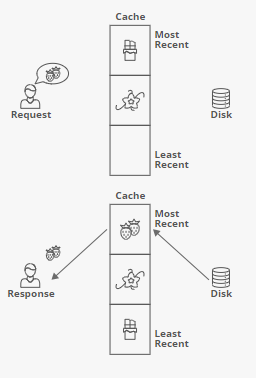


Next, someone requests the vanilla cake recipe:

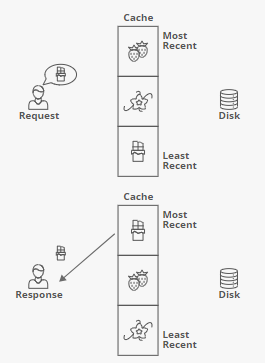


Notice that the chocolate cake recipe got bumped down a level in the cache - it's not the most recently used anymore.

Next comes a request for the strawberry cake recipe:

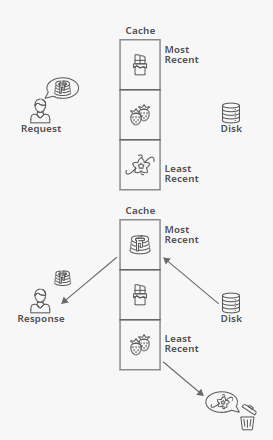


And one for chocolate:



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:



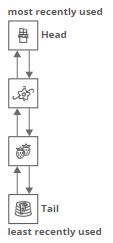
Since our cache could only hold three recipes, we had to kick something out to make room. We got rid of **("evicted")** the vanilla cake recipe, since it had been used least recently of all the recipes in the cache. This is called a **"Least-Recently Used (LRU)" eviction strategy.**

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) 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:

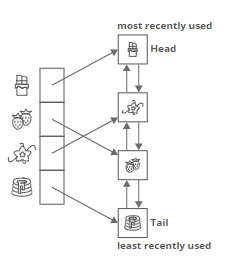


This lets us access the LRU element in O (1) O (1) time by looking at the tail of the list.

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)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:



**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"
  + Use the hash table to quickly find the corresponding linked list node.
  + 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:
  + 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.
  + Create a new linked list node for the item. Insert it at the head of the linked list.
  + 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) O (1), so put together it takes O (1) O (1) time to update our cache each time an element is accessed. Pretty cool!