Hash table

Theory Practice 100% completed, 0 problems solved

Theory

11 minutes reading

Unskip this topic Start practicing

2 required topics

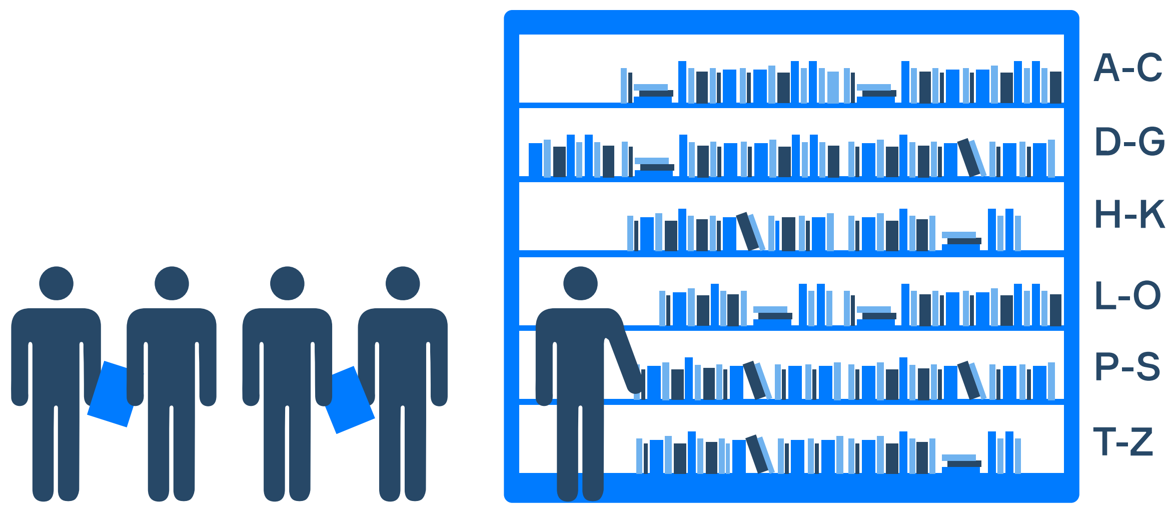
✓ Hash function
✓ Data structures

7 dependent topics

✓ The Map interface
HashMap
hashCode() and equals()
Maps
Hashable
Collisions
Memory-efficient data structures in Android

§1. Introduction

Let's imagine the following scenario: you have a lot of friends and a big shelf full of books. Some of your friends want to borrow your books, some want to return them, and some want to know if you have a certain book. So, you want to write a program that lets you *add* books, *remove* books, and *check* if a book is available. For this scenario, what would be the best data structure to use?



Hash tables are structures that allow us to insert and remove values, and check if a value is present in time $O(1)$ for each of those operations. Hash tables alone cannot guarantee this time. However, paired with a good hash function, everything works well, making it one of the best data structures for this purpose.

Table of contents:

§2. Hash tables

Hash tables are arrays where each entry is a **bucket**. Buckets can hold 0 or more values of a type. They are identified by their index in the array. If we want to insert a value in the hash table, we compute its hash value and insert it in the bucket at an index equal to the hash value (modulo the size of the array if the hash value is too large). You can do the same to remove an object or search for it.

Let's look at an example now. Say we have a hash table with 5 buckets, and we are inserting integers {1, 3, 5, 6} with the identity hash function. The table looks like this:

Index	0	1	2	3	4
Values	{5}	{1, 6}	{}	{3}	{}

Now let's figure out how it works. Since we're using the identity function, the hash value is equal to the value itself. Because of this, you can see that 1 and 3 are in buckets at indexes 1 and 3 respectively. Now, 5 and 6 have hash values 5 and 6, but there are too few buckets! To find a bucket for them, we take the modulo of the hash value by the total number of buckets, in this case, 5. So, 5 modulo 5 is 0, and 6 modulo 5 is 1, and you can see in the table that the values are placed in the correct buckets.

There are two types of hash tables. The one above is called a **hash set**. We can use it in scenarios where we only need to check if a value is present: for example, the book scenario in the introduction. We can still add or remove elements to a hash set, but in most cases we are not interested in getting a value out of a hash set. This happens because we already have an equal value that we need in order to search a hash table. Implementations of hash sets are *unordered_set* in C++ and *HashSet* in Java.

Another type of hash table is a **hash map**. Imagine you want to keep a phone book of names and numbers of friends, and want to search for phone numbers using a friend's name. If you kept a hash set of the name-number pairs, you would need to know the number to be able to search. Here, hash maps can help you. They are very similar to hash sets, but they store pairs. The first entry in the pair is called the *key*, and the second is the *value*. Only the hash of the key is used, and, when searching, we look for the value associated with a key. In the example above, the keys would be the names, and the values would be the phone numbers. Implementations of hash maps are *unordered_map* in C++ and *HashMap* in Java.

§3. Load factor

Having huge buckets is always bad news! Imagine a hash table with one bucket and a huge number of elements. Then every time we want to do something, we have to search through the whole bucket to find the elements we need. That's the same thing as putting all the elements into an array without any order! If we allowed this to happen, what would be the point of using a hash table?

Now, think about a more common example: a hash table with 100 buckets and 200 values in it. An ideal hash function would spread the values uniformly, and we would have 2 values in each bucket. Then, whenever we check for a value, we have to check for equality with both values in the corresponding bucket. This is not bad, but, ideally, buckets should have 1 or 0 elements. We can't always guarantee it, but we can improve the average number if, for example, we have more buckets than elements. For this, we have to introduce the load factor.

The **load factor** of a hash table is a real number l between 0 and 1 that tells us how full the hash table is. We can calculate it at any time with this formula:

$l = \frac{\#elements}{\#buckets}$.

Most hash tables have a **max load factor** α, a constant number between 0 and 1 that is an upper limit for the load factor. After we insert a new value to the hash table, we calculate the new load factor l. If $l > \alpha$, then we increase the number of buckets in the hash table, usually by creating a new array of twice as many buckets and inserting in it all the values from the old one. Note that we have to recompute the indices of all elements, since their indices are based on both hash values and the total number of buckets. A common value for the max load factor is 0.75, which helps us make sure there are enough buckets, while not wasting a lot of memory on empty ones.

This picture illustrates that the emptier the bucket, the better it is for the performance of the hash table. The load factor helps us keep the buckets as close to empty as reasonably possible. A too low load factor, however, may also be a bad sign, as it means that we use a lot of unnecessary memory to store empty buckets.

§4. Why are hash tables so fast?

Earlier we mentioned that hash tables take $O(1)$ to insert, remove, or search for a value when using a good hash function. Let's see why this is true!

For all those operations, the hash table has to do exactly 2 things: compute the hash value and search in only one bucket for the initial value, namely in the bucket whose index is equal to the computed hash value. A good hash function is *efficient*, so it takes $O(1)$ to compute the hash value and is *deterministic*, so the hash value will be the same for any equal values, meaning that we will search the correct bucket. Next, a good hash function is *uniform*, so there will not be some very large buckets, while others are empty or almost empty. This, together with the load factor we explained above, makes sure that, on average, there is no more than one element in a bucket. So, searching the bucket takes time $O(1)$. Finally, we can implement buckets so that inserting and deleting from them also takes $O(1)$, for example, using linked lists. So, if we are using a good hash function, all operations on a hash table will take $O(1)$.

§5. Summary

- Hash tables are data structures that support fast inserting and removing values, and checking if a value is present.
- Hash tables consist of buckets holding zero, one, or several values.
- Hash functions are used to determine a bucket for a value.
- Hash sets store objects based on their hash values; hash maps store key-value pairs based on the keys' hash values.
- Load factor is typically used to determine when to resize the hash table to keep it fast.