Chapter 5

Hash Tables

In this chapter we will learn all about hash tables, one of the most important data structures, we will see its use cases, and its internals (implementation, collision and hash functions).

Suppose that you are working in a supermarket, when a customer asks you about the price of a certain product, you must search for that product in the unalphabetized prices book which will take O(n), very consuming right? What if the book was alphabetized? It will be much faster and will take only O(Log(n)) but still the waiting customer will not be very happy about it, so you decide to memorize the whole book, now when you are asked you answer instantly in O(1) its even faster and better than binary search, now in terms of computer science you have an array, each index contains the product name and its size , if it's not sorted it will take O(n), if it is sorted by name it will take O(Log(n)) , but if you want to retrieve it in O(1) you need hash tables.

Hash Functions:

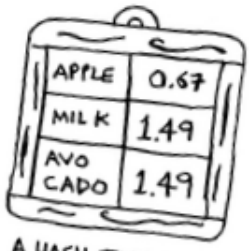
A hash function is a function where you put in a string, and you get back a number. We can say that it maps strings to numbers, this number will represent the index where your key and value are saved, and it has some requirements:

- Consistency: every time you enter a key (the name of the product) the hash function maps it to the same number, so if apple is mapped to 3 it will be saved in index number 3, consistency is important so that we can retrieve the value(price).
- Different words should be mapped to different numbers.
- Hash function should always return a valid index.

AN EMPTY HASH TABLE

book is a new hash table. Let's add some prices to book:

```
>>> book["apple"] = 0.67   <-- An apple costs 67 cents.
>>> book["milk"] = 1.49    <-- Milk costs $1.49.
>>> book["avocado"] = 1.49
>>> print book
{'avocado': 1.49, 'apple': 0.67, 'milk': 1.49}
```



A HASH TABLE OF PRODUCE PRICES

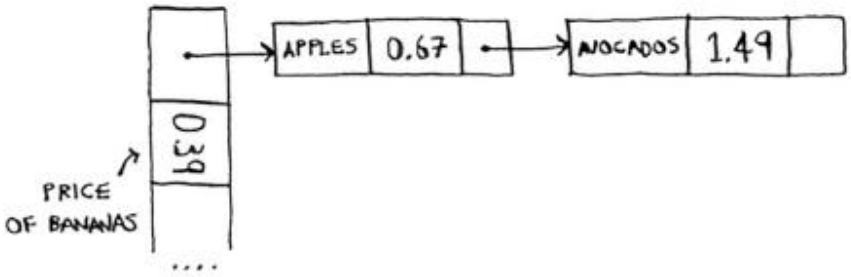
Use Cases:

- Lookups: for example, your phone contacts, when you want to add a new contact, you enter the person's name(key) then his phone number(value), and when you want to get the number, you can easily get it by searching the name.
- DNS resolution: taking a web address and translate it into IP address.
- Preventing duplicates: in hash tables the key must be unique, you can test if this key exists by checking if it has a value or not.

- Cashing: on a website if there is a webpage that often requested, and it doesn't change from one user to another (like the login page) instead of making the server handle the request of going to that page you can use hash tables for faster response.

Collisions:

We said that hash keys map different keys to different numeric values but that's just an ideal case, in real life it is impossible to have a hash function to do that. For example, let's say we have array of size 26 and the function assigns a spot in the array alphabetically. First, we add apple and its price without a problem but what if we add avocado? Now avocado will replace apple and its price and that's what we call collision. There are many ways to deal with collisions. The simplest one is this: if multiple keys map to the same slot, start a linked list at that slot.



This way when you look for bananas it's fast but if you look for apples or avocadoes it is a bit slower, it's ok for small number of items, but what if we have a lot of products begin with letter a? That will really slow us down, what if all the products begin with letter a? the whole hash table will be empty except the first index.
Hash functions must map keys evenly all over the hash. And must not allow the linked list to get too long.

Performance:

	HASH TABLES (AVERAGE)	HASH TABLES (WORST)	ARRAYS	LINKED LISTS
SEARCH	O(1)	O(n)	O(1)	O(n)
INSERT	O(1)	O(n)	O(n)	O(1)
DELETE	O(1)	O(n)	O(n)	O(1)

in the average case for hash tables. Hash tables are as fast as arrays at searching (getting a value at an index). And they're as fast as linked lists at inserts and deletes. It's the best of both, but in the worst case, hash tables are slow at all of those. So, it's important that you don't hit worst-case performance with hash tables. And to do that, you need to avoid collisions. To avoid collisions, you need

- A low load factors
- A good hash functions

Loading Factor:

Loading factor = number of occupied slots in an array/ number of total slots.

- If it is less than 1 then there are still empty slots.
- If it is one or more then the array is full, and we must resize.

Resizing: you create a new array that's bigger. Usually, we make an array that is twice the size. Then we re-insert all those items into this new hash table using the hash function.
It is best practice to resize when the loading factor is 0.7.