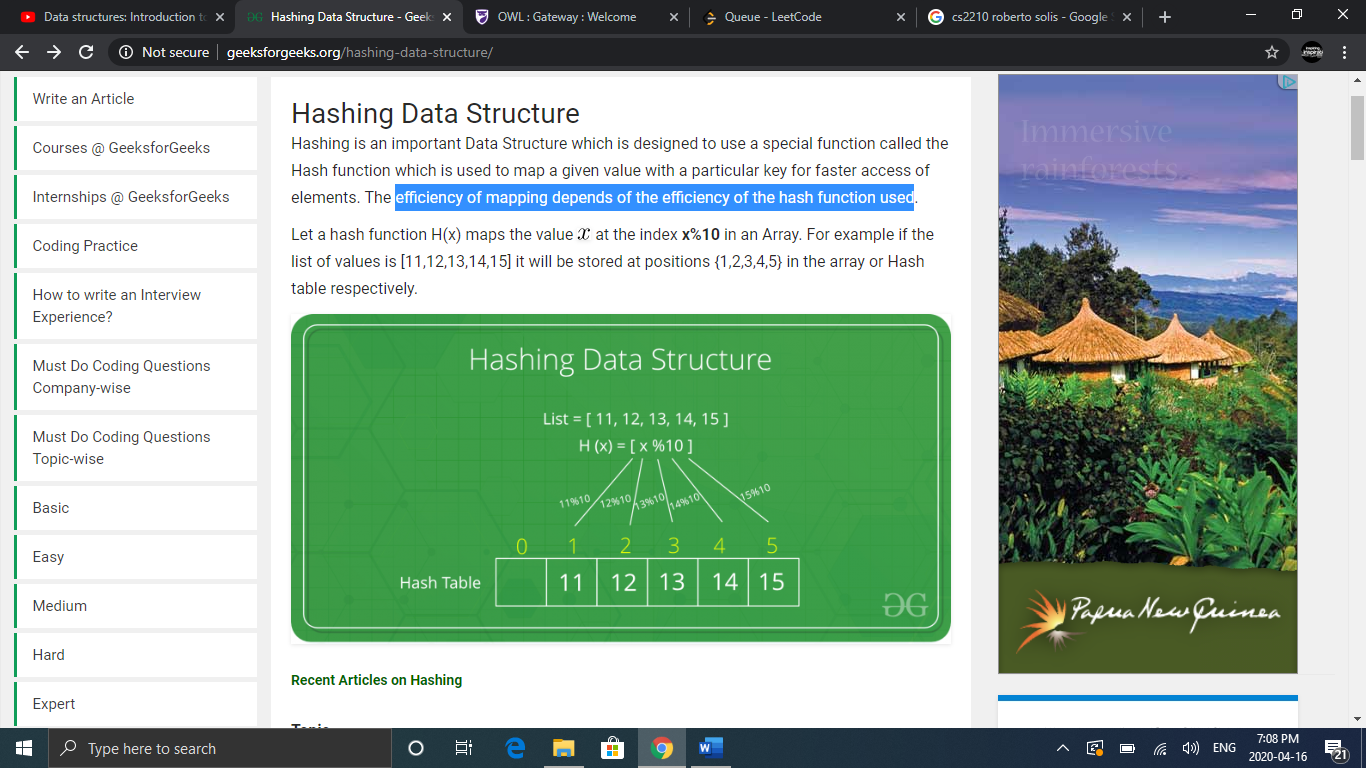
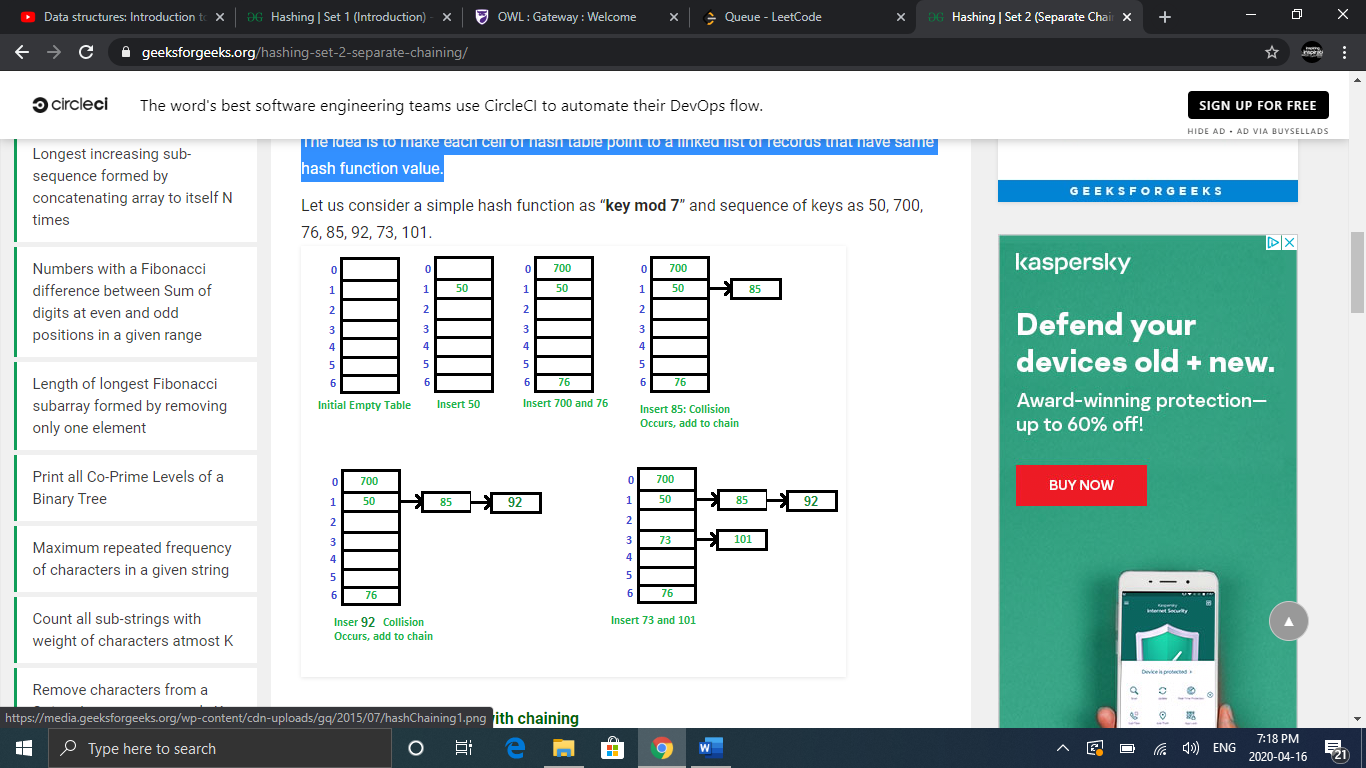
Hash Tables

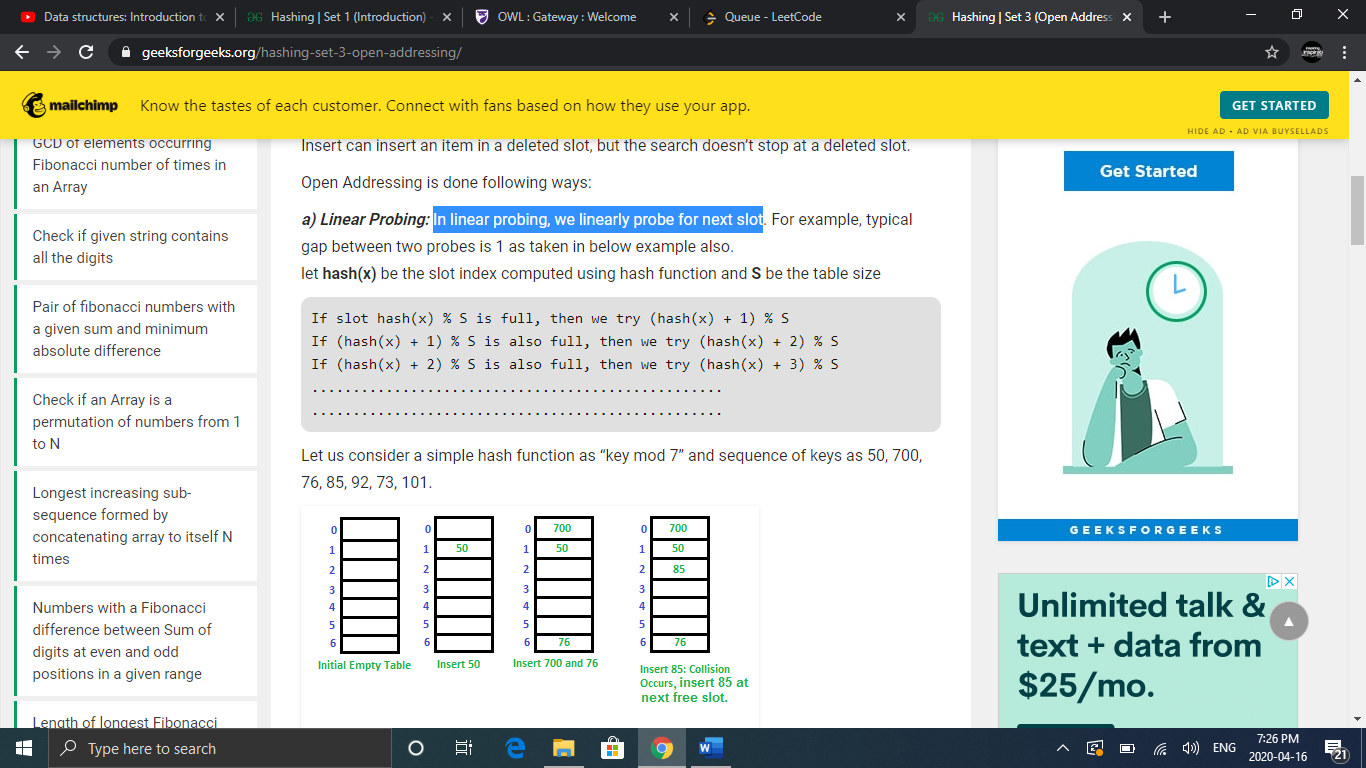
* designed to use a special function called the Hash function which is used to map a given value with a particular key for faster access of elements
* efficiency of mapping depends of the efficiency of the hash function used
* we use a hash function to map values at x
* A hash function is simply x%m where m is a positive integer
* A good hash function should have following properties:
* 1) Efficiently computable.
* 2) Should uniformly distribute the keys (Each table position equally likely for each key)



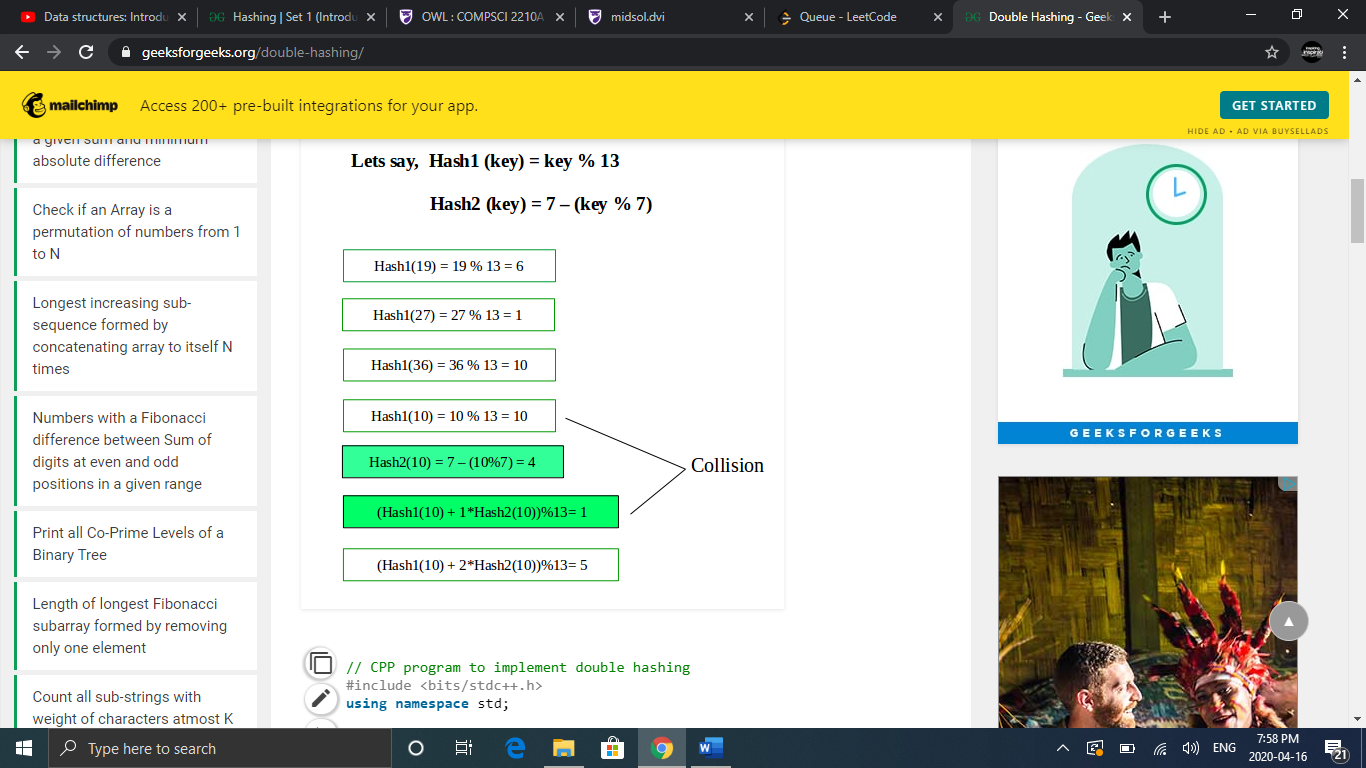
* There is a possibility that a new key is mapped to an occupied position in the hash table. This is called a collision
* There are different ways to handle collisions handling technique:
* 1) Separate Chaining: each cell points to a collection of values that have the same value when computed in the hash function
* Ex. Suppose our hash function is key mod 7 (H(x)= x%7)



* Spereate chaining is advantageous because the hash table never fills up, we can always add more links, however more links uses more memory
* It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.
* 2) Open Addressing: a) Linear Probing
* In linear probing, we linearly search for next empty slot (slot index+1)%m
* We will look at the same example used above



* The issue with this method is that many consecutive elements get clustered and it begins taking time to searching for a free slot or for an element
* 2) Open Addressing: b)Double Hashing
* This method uses a second hash function when a collision occurs
* Ex. Hash1(key) =key% 13 and Hash2(key) = 7-(key%7)



* Open addressing uses less memory than separate chaining
* Java has a built in method for hash tables, we will look at some examples below

# Hash Sets

* Hash Sets just have a value (no key)
* Hash Sets have distinct values
* Hash Sets can be used to check if something occurs
* Java Hash Set method:

        // Adding elements into HashSet using add()

        h.add("India");

        h.add("Australia");

        h.add("South Africa");

        h.add("India");// adding duplicate elements , overrides it

// Removing items from HashSet using remove()

        h.remove("Australia");

// Iterating over hash set items in java

        Iterator<String> i = h.iterator();

        while (i.hasNext()){

            System.out.println(i.next());

    }

**Output:**

South Africa

India

## **HashSet Example**

Given a **non-empty** array of integers, every element appears twice except for one. Find that single one.

class Solution {

public int singleNumber(int[] nums) {

HashSet<Integer> set = new HashSet<>();

for(int i=0;i<nums.length;i++){

if(set.contains(nums[i])){set.remove(nums[i]);}

else{set.add(nums[i]);}

}

Iterator<Integer> i = set.iterator();

return i.next();

}

}

# Hash Maps

* Have a key and a corresponding value
* Keys are distinct but values may not be
* HashMaps can be used to count how many times something occurs
* Hash Map in Java:

HashMap<Character, Integer> map = new HashMap<>();

// Key(type) Value(type)

//add values to your hashMap

map.put('A', 65);  
map.put('B', 66);

//print a set of all the keys

System.out.println(map.keySet());

**Output:**

[A,B]

## **HashMaps with Heaps**

* Hash Maps can be implemented in Java with Priority Queues (heaps)
* This feature is useful in a situation where you want to find what occurred the most often in a problem
* The HashMap will allow you to count all occurrences and the heap will allow you to quickly determine the least or highest occurrence
* Java Declararion:

PriorityQueue<Map.Entry<Character,Integer>> heap=new PriorityQueue<>((o1, o2) -> o2.getValue()-o1.getValue());

heap.addAll(map.entrySet()); //Adds all keys in the map to the heap