How hashing works:

For **insertion** of a key(K) – value(V) pair into a hash map, 2 steps are required:

1. K is converted into a small integer (called its hash code) using a hash function.
2. The hash code is used to find an index (hashCode % arrSize) and the entire linked list at that index(Separate chaining) is first searched for the presence of the K already.
3. If found, it's value is updated and if not, the K-V pair is stored as a new node in the list.

Complexity and Load Factor

- For the **first step**, the time taken depends on the K and the hash function.
  For example, if the key is a string "abcd", then it's hash function may depend on the length of the string. But for very large values of n, the number of entries into the map, and length of the keys is almost negligible in comparison to n so hash computation can be considered to take place in constant time, i.e, **O(1)**.
- For the **second step**, traversal of the list of K-V pairs present at that index needs to be done. For this, the worst case may be that all the n entries are at the same index. So, time complexity would be **O(n)**. But, enough research has been done to make hash functions uniformly distribute the keys in the array so this almost never happens.
- So, **on an average**, if there are n entries and b is the size of the array there would be n/b entries on each index. This value n/b is called the **load factor** that represents the load that is there on our map.
- This Load Factor needs to be kept low, so that number of entries at one index is less and so is the complexity almost constant, i.e., O(1).

Rehashing:

Rehashing is the process of increasing the size of a hashmap and redistributing the elements to new buckets based on their new hash values. It is done to improve the performance of the hashmap and to prevent collisions caused by a high load factor.

When a hashmap becomes full, the load factor (i.e., the ratio of the number of elements to the number of buckets) increases. As the load factor increases, the number of collisions also increases, which can lead to poor performance. To avoid this, the hashmap can be resized and the elements can be rehashed to new buckets, which decreases the load factor and reduces the number of collisions.

During rehashing, all elements of the hashmap are iterated and their new bucket positions are calculated using the new hash function that corresponds to the new size of the hashmap. This process can be time-consuming but it is necessary to maintain the efficiency of the hashmap.

Why rehashing?
Rehashing is needed in a hashmap to prevent collision and to maintain the efficiency of the data structure.
As elements are inserted into a hashmap, the load factor (i.e., the ratio of the number of elements to the number of buckets) increases. If the load factor exceeds a certain threshold (often set to 0.75), the hashmap becomes inefficient as the number of collisions increases. To avoid this, the hashmap can be resized and the elements can be rehashed to new buckets, which decreases the load factor and reduces the number of collisions. This process is known as rehashing.

Rehashing can be costly in terms of time and space, but it is necessary to maintain the efficiency of the hashmap.

How Rehashing is done?
Rehashing can be done as follows:

- For each addition of a new entry to the map, check the load factor.
- If it’s greater than its pre-defined value (or default value of 0.75 if not given), then Rehash.
- For Rehash, make a new array of double the previous size and make it the new bucketarray.
- Then traverse to each element in the old bucketArray and call the insert() for each so as to insert it into the new larger bucket array.