A HashMap is a data structure that stores key-value pairs and allows for efficient retrieval, insertion, and deletion of elements based on their keys. It is also known as a dictionary, associative array, or hash table in other programming languages.

Here are some key points about HashMaps:

- Key-Value Pairs: Each element in a HashMap is a combination of a key and a value. The key is used to uniquely identify the associated value.
- Hashing: HashMaps internally use a hashing function to map keys to indices in an array. This allows for fast retrieval and insertion of elements. When you want to access a value associated with a key, the hashing function computes the index where the value is stored, enabling constant-time (O(1)) access in many cases.
- Fast Operations: HashMaps typically provide fast operations for insertion, deletion, and retrieval of elements. These operations usually have an average-case time complexity of O(1) when the hashing function distributes the keys evenly across the array.
- Unordered Collection: HashMaps do not guarantee any particular order of elements. The order in which elements are stored and retrieved may not be consistent.
- No Duplicate Keys: Each key in a HashMap must be unique. If you attempt to insert a key-value pair with a key that already exists in the HashMap, the previous value associated with that key will be overwritten.

When to use a HashMap:

- Fast Lookup: Use HashMaps when you need to store key-value pairs and require fast lookup operations based on keys. HashMaps provide constant-time access to elements on average, making them suitable for scenarios where you frequently need to retrieve values based on keys.
- Uniqueness: HashMaps are useful when you need to ensure that keys are unique within the collection. They automatically handle cases where duplicate keys are attempted to be inserted.
- Associative Data: If you have data that naturally fits into a key-value structure, such as mapping usernames to user information or tracking word frequencies in a document, HashMaps are an excellent choice.
- Efficient Inserts and Deletes: If you need to frequently insert or delete elements while maintaining fast access times, HashMaps are a good option. They provide efficient operations for adding and removing key-value pairs.

Overall, HashMaps are versatile data structures that are commonly used in various programming scenarios where fast access to key-value pairs is required. They are particularly useful for implementing dictionaries, caches, and indexing structures.

Unique number of occurrences (GeeksforGeeks problem): Given an array arr of N integers, the task is to check whether the frequency of the elements in the array is unique or not. Or in other words, there are no two distinct numbers in array with equal frequency. If all the frequency is unique then return true, else return false.
Your task:
You don't need to read input or print anything. Your task is to complete the function isFrequencyUnique() which take integer N and array arr of size N as arguments, and returns a boolean.

Expected Time Complexity: O(N)
Expected Auxiliary Space: O(N)

Constraints:
1 <= N <=105
-109 <= arr[i] <= 109

This Java function first creates a HashMap `frequencyMap` to store the frequency of each element in the array. Then, it creates another HashMap `frequencyCount` to store the counts of unique frequencies. Finally, it checks if any frequency count is greater than 1. If so, it returns `false` indicating that the frequencies are not unique. Otherwise, it returns `true`.
Here's an explanation of the solution:

Counting Frequency of Each Element:

- First, we create a `HashMap` called `frequencyMap` to store the frequency of each element in the input array.
- We iterate through the array `arr` and update the count for each element in the map. If an element is already present in the map, we increment its count by 1. If it's not present, we initialize its count to 1.

Counting Frequency Counts:

- After counting the frequency of each element, we need to count how many times each unique frequency appears. We create another `HashMap` called `frequencyCount` for this purpose.

- We iterate through the values (frequencies) in the `frequencyMap` and update the count of unique frequencies in the `frequencyCount` map. Again, if a frequency count is already present, we increment its count by 1. If it's not present, we initialize its count to 1.

Checking for Uniqueness:

- Finally, we iterate through the values (frequency counts) in the `frequencyCount` map.
- If any frequency count is greater than 1, it means there are multiple elements with the same frequency, violating the uniqueness condition. In this case, we return `false`.
- If all frequency counts are 1 (each frequency appears only once), then we return `true`, indicating that all frequencies are unique.

Time and Space Complexity:

- The time complexity of this solution is O(N), where N is the size of the input array. This is because we iterate through the array only once to count frequencies.
- The space complexity is also O(N) because we use two HashMaps, `frequencyMap` and `frequencyCount`, each potentially containing up to N entries, depending on the uniqueness of elements and their frequencies.

Overall, this solution efficiently counts the frequencies of elements and their frequency counts, allowing us to determine whether the frequencies are unique in linear time.