1. What is a HashMap?

A **HashMap** is a data structure that stores data in **key-value pairs**. It uses a **hashing technique** to compute the index of the bucket where the value will be stored.
In **Java**, it's part of the java.util package:

```
HashMap<KeyType, ValueType> map = new HashMap<>();
```

2. Internal Working of HashMap (High-Level)

- **Key** is passed to a **hash function** which returns a hash code.
- This code is then **modulo'd with the capacity** to find the index (bucket).
- In case of **collision** (multiple keys map to same index), **chaining (linked list)** or **tree (for high collisions)** is used.
- Lookup and insertion generally take **O(1)** time.

3. Where Can HashMap Be Applied?

HashMaps are extremely powerful when:

You need fast lookups (O(1) average time).

You need to count frequencies.

You need to check existence of an element.

You want to store unique mappings (like ID → value).

You need caching/memoization.

4. How to Apply HashMap in Problems?

Let’s break this into **step-by-step**:

Use when:

- You see **"count", "most", "duplicate", "frequency", "pair", "index lookup", "group by"** in the problem.
- You need to map **one type to another** (e.g., String → Integer).
- You need to **remember previously seen values** (for optimization or fast lookup).

5. Example Use Cases with Code

1. Count Frequency of Characters

```
Map<Character, Integer> freq = new HashMap<>();
for (char c : str.toCharArray()) {
   freq.put(c, freq.getOrDefault(c, 0) + 1);
}
```

2. Two Sum Problem

Find two indices such that nums[i] + nums[j] == target.

```
Map<Integer, Integer> map = new HashMap<>();
for (int i = 0; i < nums.length; i++) {
    int complement = target - nums[i];
    if (map.containsKey(complement)) {
        return new int[]{map.get(complement), i};
    }
    map.put(nums[i], i);
}
```

3. Group Anagrams

Group words that are anagrams of each other.

```
Map<String, List<String>> map = new HashMap<>();
for (String word : words) {
    char[] chars = word.toCharArray();
    Arrays.sort(chars);
    String key = new String(chars);
    map.computeIfAbsent(key, k -> new ArrayList<>()).add(word);
}
```

6. How to Recognize HashMap Pattern in DSA Questions

Hint in Problem	**Why**
"Find duplicates"	Track seen values.
"Find frequency/count"	Use value as counter.
"Get index of previous value"	Store index as value.
"Fast lookups of elements"	HashMap is O(1) lookup.
"Group by value"	Key = group, Value = List.
"Check if element seen before"	Use containsKey.

7. Advantages of Using HashMap

Advantage	**Explanation**
O(1) time lookup	Fast get/put/remove.

Easy to **count data**	Frequency maps are straightforward.
Good for **memoization**	Store intermediate results in recursion/DP.
Can map **any type**	Supports generic key-value.
Useful in **backtracking/DFS**	To avoid recomputation.

8. Common LeetCode Questions Using HashMap

Problem	**Why HashMap?**
Two Sum	Index tracking
Group Anagrams	Group by sorted key
Longest Substring Without Repeating	Store last seen index
Top K Frequent Elements	Count frequency
Subarray Sum Equals K	Prefix sum tracking

9. Best Practices

- Use getOrDefault() for counting.
- For List values, use computeIfAbsent() to simplify code.
- Be careful when using **mutable objects** as keys (they can change their hash).

10. Summary

Feature	**HashMap**
Access time	O(1) avg
Stores	Key-Value
Null allowed?	1 null key, many null values
Ordered?	(Use LinkedHashMap for order)
Thread-safe?	(Use ConcurrentHashMap for that)

INTERNAL WORKING

Overview

A HashMap in Java stores key-value pairs and provides **O(1) average time complexity** for get() and put() operations using **hashing**.
It works like:

```
Map<String, Integer> map = new HashMap<>();
map.put("Karan", 25);
```

2. Internal Data Structure

Internally, a HashMap uses:

- **An array of Node<K,V>** (or Entry): Each array index is a **bucket**
- Each node contains:
    - hash (hash code of key)
    - key
    - value
    - next (for chaining in case of collisions)

```
static class Node<K,V> implements Map.Entry<K,V> {
   final int hash;
   final K key;
   V value;
   Node<K,V> next;
}
```

3. How put(key, value) Works

Step-by-step:

1. **Compute hash** of the key:

    ```
    hash = hash(key.hashCode());
    ```

1. **Index Calculation**:

    ```
    index = (n - 1) & hash; // where n = capacity of table
    ```

1. **Insert node**:

    - **If no collision (bucket is empty): Add node directly.**
    - **If collision (same bucket already has a node):**
        - **Traverse linked list to check if key already exists:**
            - **If yes: overwrite the value.**
            - **If no: append node at the end of the chain.**

4. How get(key) Works

1. Compute hash and index like in put.
2. Traverse the bucket (linked list or tree):
    - If key found, return value.
    - Else, return null.

Example

```
Map<String, String> map = new HashMap<>();
map.put("A", "Apple");
map.put("B", "Banana");
map.put("A", "Avocado"); // Overwrites value for key "A"
```

Behind the scenes:

- "A".hashCode() → index
- Stored in bucket
- When "A" is inserted again, it finds same key, updates value

Time Complexity

Operation	Average Case	Worst Case
put()	O(1)	O(log n) (tree) or O(n) (linked list)
get()	O(1)	O(log n) or O(n)
remove()	O(1)	O(log n) or O(n)

Summary

Concept	Details
Storage	Array of Node<K,V> (buckets)

Collision	Handled by chaining (list/tree)
Hash function	(h = key.hashCode()) ^ (h >>> 16)
Resize	When size > threshold (capacity * load factor)
Treeify	Linked list → Red-Black Tree if collisions > 8
Null keys/values	One null key, multiple null values allowed

#	Problem Title	Pattern Use	Link
1	Two Sum	Value-to-index map	🔗 Link
2	Group Anagrams	Key = sorted string, Value = List	🔗 Link
3	Longest Substring Without Repeating Characters	Char-to-index map	🔗 Link
4	Subarray Sum Equals K	Prefix sum to count map	🔗 Link
5	Top K Frequent Elements	Frequency count	🔗 Link
6	Isomorphic Strings	Char mapping both directions	🔗 Link
7	Word Pattern	Index/character mapping	🔗 Link
8	Find All	Char frequency	🔗 Link

	Anagrams in a String	sliding window	
9	Longest Palindrome by Concatenating Two Letter Words	Word frequency pairing	🔗 Link
10	Contains Duplicate	HashSet check for duplicates	🔗 Link
11	Valid Anagram	Char frequency map	🔗 Link
12	Minimum Window Substring	Sliding window with char count map	🔗 Link
13	Binary Tree Vertical Order Traversal	Column index → node list map	🔗 Link
14	Find Duplicate Subtrees	Subtree serialization map	🔗 Link
15	Longest Consecutive Sequence	HashSet + map for sequence start	🔗 Link