**Definition/Concept**

* **HashMap (Java)**:
  + Stores key-value pairs.
  + Uses **hashing** internally to place keys into "buckets".
  + Provides **average O(1)** time for insertion, lookup, and deletion.
  + Does **not guarantee any order** of keys.

**Syntax in Java:**  
  
import java.util.HashMap;

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

map.put("apple", 3);

map.put("banana", 5);

System.out.println(map.get("apple")); // 3

## System.out.println(map.containsKey("banana")); // true ****Memory Allocation****

* Internally, HashMap uses:
  + An **array of buckets** (initially size 16).
  + Each bucket holds a linked list (or a tree if collisions get large enough).
  + Keys are hashed into an index.

So memory = array overhead + linked lists/tree nodes for collisions.

## ****Theoretical Corner Cases****

* **Order not guaranteed**: Iteration order may look random and can even change after rehashing.
* **Collisions**: Too many collisions degrade performance to O(n).
* **Load factor**: When 75% full, HashMap resizes (rehashes). Resizing is costly.
* **Nulls**: HashMap allows one null key and multiple null values. (C++ unordered\_map doesn’t).

## ****Real-Time Applications****

* Fast lookups:
  + Counting frequencies (word count, character count).
  + Implementing caches.
  + Symbol tables in compilers.
  + Associating metadata with objects (e.g., mapping userID → profile).

| **Feature** | **HashMap** | **LinkedHashMap** | **TreeMap** |
| --- | --- | --- | --- |
| **Definition** | Stores key-value pairs in a hash table (unordered). | HashMap with a **linked list** to maintain insertion order. | Stores key-value pairs in a **Red-Black Tree** (sorted order). |
| **Ordering** | No guarantee (random order). | Maintains **insertion order**. | Maintains **sorted order of keys** (natural/comparator). |
| **Performance** | O(1) average for put/get. | O(1) average for put/get. | O(log n) for put/get because of tree structure. |
| **Null Keys** | 1 null key allowed, many null values. | 1 null key allowed, many null values. | **Null key not allowed**, but null values allowed. |
| **Use Case** | When fast lookups are needed without order. | When order of insertion matters. | When you need keys sorted automatically. |
| **Syntax** | HashMap<K,V> map = new HashMap<>(); | LinkedHashMap<K,V> map = new LinkedHashMap<>(); | TreeMap<K,V> map = new TreeMap<>(); |

### Time Complexity Table: HashMap vs TreeMap:

| **Operation** | **HashMap** | **TreeMap** |
| --- | --- | --- |
| **Insertion** | Best/Average: **O(1)** Worst: **O(n)** (all keys collide) | Best/Average/Worst: **O(log n)** (Red-Black Tree balancing) |
| **Lookup / Get** | Best/Average: **O(1)** Worst: **O(n)** (all keys collide) | Best/Average/Worst: **O(log n)** |
| **Deletion** | Best/Average: **O(1)** Worst: **O(n)** | Best/Average/Worst: **O(log n)** |
| **Iteration** | O(n) (order not guaranteed) | O(n) (keys sorted) |
| **Memory Overhead** | Moderate (buckets + linked lists/trees in collisions) | Higher (tree nodes + pointers) |

### Key Notes:

1. **HashMap**
   * Uses **hash table** internally.
   * Average case is **O(1)** because hash function spreads keys uniformly.
   * Worst case occurs when **all keys hash to the same bucket**, forming a linked list → O(n).
2. **TreeMap**
   * Uses **Red-Black Tree** internally.
   * Always **O(log n)** for insert, get, and delete, because of tree height.
   * Keys are **always sorted**, so iteration gives sorted order.
3. **Best Case Example (HashMap)**
   * Keys spread evenly → O(1) insert/get per key.
4. **Worst Case Example (HashMap)**
   * All keys collide → single bucket → behaves like linked list → O(n).
5. **TreeMap** doesn’t have hash collisions — it always maintains balance → predictable **O(log n)**.