**Can objects be keys in the hashmaps**

Ideally, the objects used as keys in a HashMap should be immutable. This is because the key’s hash code is used for indexing, and if the key’s content changes after being used in the HashMap, the hash code could change, which would break the map’s internal structure.

Lists are mutable (i.e., their contents can change), so you should be cautious when using a mutable object like List<Student> as a key. For example, if you modify the list after it has been added to the HashMap, it could result in unexpected behavior or errors because the hash code might change, causing the map to behave incorrectly.

**Key Concepts of HashMap:**

1. **Buckets**:
   * **HashMap** internally uses an array of **buckets** (an array of **LinkedLists** or **Red-Black Trees** in some cases for high collision rates).
   * Each element in the map is stored in a bucket based on the **hash code** of the key.
2. **Hashing**:
   * When a key-value pair is added to the **HashMap**, the key is hashed using the key's hashCode() method.
   * This hash code is then **mapped to an index** in the array of buckets using a formula, such as:

index = hashCode % numberOfBuckets

* + This index determines which bucket (or LinkedList) the key-value pair will be placed in.

1. **Collision Handling**:
   * **Collisions** occur when multiple keys produce the same hash code and thus map to the same bucket.
   * Java's **HashMap** handles collisions using **chaining**:
     + If two keys hash to the same index, they are stored in a **LinkedList** at that bucket index.
     + Starting from **Java 8**, if a bucket exceeds a certain threshold (e.g., 8 elements in a bucket), the bucket may be converted into a **Red-Black Tree** for better performance (O(log n) instead of O(n) for searching).

Example of collision handling (with a LinkedList):

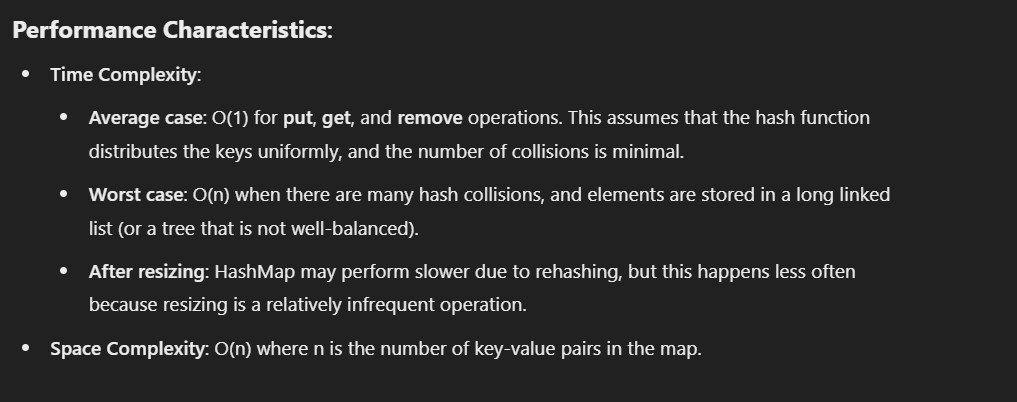
rust

Copy

Bucket 0: [Key1 -> Value1, Key2 -> Value2]

Bucket 1: [Key3 -> Value3]

1. **Resizing**:
   * A **HashMap** dynamically resizes when the number of elements exceeds a certain threshold (i.e., the **load factor**).
   * By default, the **initial capacity** is 16 and the **load factor** is 0.75.
   * This means the map will **resize** when it reaches 12 entries (16 \* 0.75).
   * When the map resizes, the number of buckets is doubled, and all the entries are rehashed (moved to new buckets). This operation is **expensive** because it involves recalculating hash values and redistributing entries.



A **HashSet** in Java is backed by a **HashMap** where each element is stored as a key, and the value is typically a constant (e.g., Boolean.TRUE).

**Key Points:**

1. **Hashing**: Elements are hashed using their hashCode() to determine their position in the internal array (buckets).
2. **No Duplicates**: It prevents duplicate elements because each element is treated as a unique key.
3. **Performance**: Average time complexity for operations like add, remove, and contains is **O(1)**, but can degrade with many collisions.
4. **Resizing**: The internal array grows when the set exceeds a threshold, ensuring a balanced hash distribution.
5. **Unordered**: Elements are stored in no specific order.