The internal working of **HashMap** in Java involves several key components and mechanisms that optimize it for fast data storage and retrieval.

### 1. ****HashMap Basics****

* **Structure**: HashMap is an array of buckets. Each bucket is a linked list or, in Java 8+, a balanced tree for handling collisions.
* **Storage**: HashMap stores data as **key-value pairs**. The keys are unique, and the values can be duplicates.

### 2. ****How Data is Stored****

1. **Key-Value Pairs**: Each key-value pair is stored in an object of the Node class.

static class Node<K, V> {

final int hash;

final K key;

V value;

Node<K, V> next; // For handling collisions

}

**2. Hash Function**: HashMap calculates a hash value for the key using the hashCode() method of the key object.

* The hash is further processed to determine the **bucket index**:

int hash = key.hashCode();

int bucketIndex = (hash & 0x7FFFFFFF) % capacity;

This ensures the hash value maps within the range of bucket indices.

### 3. ****How Data is Retrieved****

1. **Hashing**: When retrieving a value with get(key), HashMap recalculates the hash and determines the bucket index.
2. **Bucket Search**: Within the bucket, it uses the equals() method to find the correct key-value pair.

### 4. ****Collision Handling****

A collision occurs when two keys generate the same bucket index. HashMap handles collisions using:

1. **Separate Chaining**:
   * Keys with the same hash index are stored in a linked list within the same bucket.
   * As the bucket grows, it degrades performance from O(1) to O(n) in worst cases (pre-Java 8).
2. **Treeify (Java 8 and later)**:
   * If a bucket contains more than 8 nodes (default threshold), the linked list is converted into a balanced tree (e.g., Red-Black Tree).
   * This improves lookup performance to O(log n) instead of O(n).

### 5. ****Rehashing****

When the number of elements exceeds the **load factor** (default is 0.75), the HashMap resizes itself:

1. A new bucket array with double the capacity is created.
2. Existing key-value pairs are **rehashed** and redistributed into the new array.
   * This process is computationally expensive, so it is optimized to occur only when necessary.

### 6. ****Methods Involved****

1. put() **Operation**:
   * Calculates the hash and determines the bucket index.
   * If the key is already present, it updates the value.
   * Handles collisions by appending to the linked list or tree.

Example:

map.put("A", 1); // Insert

map.put("A", 2); // Updates value for "A"

2. get() **Operation**:

* Retrieves the bucket index.
* Searches the bucket using equals() to find the key and return the value.

int value = map.get("A"); // Retrieves the value associated with "A"

### 7. ****Key Methods in HashMap****

* hashCode():
  + Must be implemented properly for user-defined keys to avoid excessive collisions.
  + Example:

@Override

public int hashCode() {

return Objects.hash(field1, field2);

}

equals():

* Ensures correct key matching during bucket traversal.

Example:

@Override

public boolean equals(Object obj) {

if (this == obj) return true;

if (obj == null || getClass() != obj.getClass()) return false;

MyClass myClass = (MyClass) obj;

return field1 == myClass.field1 && field2.equals(myClass.field2);

}

### 8. ****Advantages****

* **Fast Retrieval**: O(1) average time complexity for get() and put().
* **Dynamic Sizing**: Automatically resizes to maintain performance.

### 9. ****Disadvantages****

* **Memory Overhead**: Requires extra memory for buckets and linked lists/trees.
* **Collision Impact**: Poor hash function implementation can lead to more collisions, degrading performance.

Diagram Representation:

HashMap:

[Bucket 0] --> (Key: "A", Value: 1) -> (Key: "B", Value: 2)

[Bucket 1] --> NULL

[Bucket 2] --> (Key: "C", Value: 3)

Understanding HashMap internals is crucial for writing efficient and bug-free code.

### ****1. HashMap Structure in Depth****

#### ****Bucket Array****

* A HashMap internally maintains an array of Node objects, which are linked lists or trees depending on the collision handling.
* Initially, the size of the bucket array is **16** (default), and this size grows when resizing occurs.
* The bucket is indexed by the hash of the key.

#### ****Node (Entry) Structure****

Each bucket holds a chain of Node objects. Each Node has:

1. hash: The hash value of the key.
2. key: The key object.
3. value: The value associated with the key.
4. next: A reference to the next Node in the chain (if any).

static class Node<K, V> {

final int hash;

final K key;

V value;

Node<K, V> next; // For collision handling

}

### ****2. Hashing Mechanism****

HashMap uses a **two-step process** to determine the bucket for a key:

#### Step 1: Compute Hash

HashMap invokes the key's hashCode() method, which returns an integer hash value.

Example:

int hash = key.hashCode();

#### Step 2: Index Mapping

HashMap uses the following formula to map the hash to a bucket index:

index = (hash & (capacity – 1))

**Why** capacity - 1**?**

* The capacity of the HashMap (number of buckets) is always a power of 2.
* This ensures that the & operation efficiently limits the index within bounds.

#### Example

For a key K with hashCode 105 and capacity 16:

index = (105 & (16 - 1)) = (105 & 15) = 9

The key is mapped to bucket index 9.

### ****3. Collision Handling in Depth****

#### Separate Chaining with Linked List

* When two keys map to the same bucket index, the Node objects are stored as a linked list.

Example:

Bucket 3 --> [Key: "A"] -> [Key: "B"] -> [Key: "C"]

Searching for a specific key involves iterating through the list and using equals().

#### Treeify (Post Java 8)

* If the length of the linked list exceeds **8** (default threshold) and the bucket array size is greater than **64**, the linked list is converted into a **Red-Black Tree**.
* Red-Black Trees allow search and insertion in O(log n) time.

### ****4. Rehashing and Resizing****

#### ****When Does Resizing Occur?****

* Resizing occurs when the number of elements exceeds the product of capacity and load factor.
  + **Load Factor**: The fraction of the bucket array that can be filled before resizing. Default is 0.75.
  + Example: For a HashMap with a capacity of 16 and a load factor of 0.75, resizing occurs after adding 12 elements.

#### ****How Resizing Works****

1. The bucket array size is doubled.
2. Existing elements are **rehashed** and redistributed into the new bucket array.

#### ****Rehashing Formula****

During rehashing, the new bucket index is recalculated:

newIndex = hash & (newCapacity – 1)

This ensures efficient redistribution of keys.

#### ****Cost of Rehashing****

* Rehashing is computationally expensive because it involves:
  1. Iterating over all existing keys.
  2. Recomputing their hash and index.
  3. Inserting them into the new bucket array.

### ****5. Important Methods in HashMap****

#### ****1.**** put(K key, V value)

The process of adding a key-value pair:

1. Compute the hash of the key and determine the bucket index.
2. Check if the key already exists in the bucket:
   * If yes, update the value.
   * If no, create a new Node and append it to the chain/tree.
3. If the number of elements exceeds the load factor, resize the HashMap.

#### ****2.**** get(K key)

The process of retrieving a value:

1. Compute the hash of the key and determine the bucket index.
2. Traverse the bucket:
   * Use equals() to find the matching key.
   * Return the associated value.

#### ****3.**** hashCode() ****and**** equals() ****in Custom Objects****

* Custom keys must override hashCode() and equals() to ensure consistent behavior.

Example:

class CustomKey {

private int id;

private String name;

@Override

public int hashCode() {

return Objects.hash(id, name);

}

@Override

public boolean equals(Object obj) {

if (this == obj) return true;

if (obj == null || getClass() != obj.getClass()) return false;

CustomKey key = (CustomKey) obj;

return id == key.id && name.equals(key.name);

}

}

### ****6. Complexity Analysis****

| Operation | Average Case | Worst Case (Before Java 8) | Worst Case (After Java 8) |
| --- | --- | --- | --- |
| put() | O(1) | O(n) | O(log n) |
| get() | O(1) | O(n) | O(log n) |
| Resizing/Rehashing | O(n) | O(n) | O(n) |

### ****7. Common Pitfalls****

1. **Poor Hash Function**: Leads to excessive collisions, degrading performance.
2. **Mutable Keys**: Changing the state of a key after it’s added to the HashMap can make it inaccessible.

Example:

map.put(new CustomKey(1, "John"), "Value");

// Modifying the key state

key.setId(2); // Key becomes unreachable

**3. Overhead of Resizing**: Adding a large number of elements at once can trigger multiple resizes.