**How HashMap Works Internally - 2022**

* HashMap uses its [static inner class](https://medium.com/javarevisited/do-you-know-nested-and-inner-classes-in-java-cc5647f46e07)**Node<K,V>** for storing map entries (). **That means each entry in hashMap is a Node**. Internally HashMap uses a **hashCode**of the key Object and this hashCode is further used by the **hash function** to find the **index of the bucket** where the new entry can be added. The formula for calculating the index of bucket (where n is the size of an array of the bucket) is given below:

**transient** Node<K,V>[] **table**; 🡸 To create an array

**Index = hashCode(key) & (n-1); 🡸 As the HashMap is always a power of 2 in size you can use as**

**hash = (null != key) ? hash(key) : 0;**

**bucketIndex = indexFor(hash, table.length);**

**HashCode used in HashMap**

**static final int** hash(Object key) {  
 **int** h;  
 **return** (key == **null**) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
}

* HashMap uses multiple buckets and each bucket points to a **Singly Linked List** where the entries (nodes) are stored.
* Once the bucket is identified by the hash function using hashcode, then hashCode is used to check if there is already a key with the same hashCode or not in the bucket(singly linked list).
* If there already exists a key with the **same** **hashCode**, then the **equals()** method is used on the **keys**. If the equals method returns **true**, that means there is already a node with the same key and hence the value against that key is **overwritten**in the entry(node), otherwise, a new node is created and added to this Singly Linked List of that bucket.
* If there is no key with the same hashCode in the bucket found by the hash function then the new Node is added to the bucket found.

**Each Node Has the Following Structure:**

**final int hash;  
final K key;  
V value;  
Node<K,V> next**

Structure is given below

Diagram

Description automatically generated with medium confidence

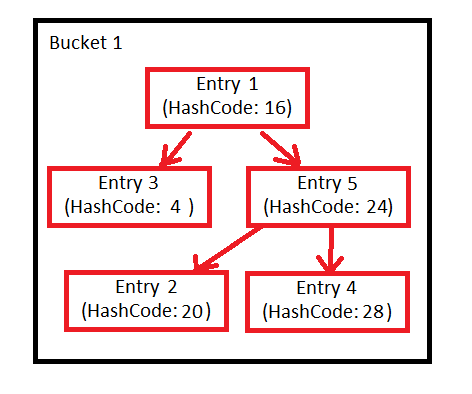
**Java 8 Changes**

**Before java 8, singly-linked lists were used for storing the nodes. But this implementation has changed to self-balancing BST after a thresold is crossed (static final int TREEIFY\_THRESHOLD = 8;).** The motive behind this change is that HashMap buckets normally use linked lists, but for the linked lists the worst-case time is O(n) for lookup. Also note that Ordinary binary search trees have pathological cases where they become O(n) [basically BST becomes skewed], but red-black/AVL trees are specifically designed to prevent these cases. In a HashMap with linked lists, if we have a really really awful hash function, we could end up with all the items hashing to the same bucket and get O(n) lookup, But it seems like with this red-black/AVL tree scheme, even if all the items hashed into the same bucket, we would get O(log⁡n) lookup in worst of worst scenario*.*

Diagram

Description automatically generated

The implementation of HashMap tries to mitigate this by organising some buckets into trees rather than linked lists if the buckets become too large. This is what TREEIFY\_THRESHOLD = 8 is for. If a bucket contains more than eight items, it should become a tree.



This tree is a [Red-Black tree](https://en.wikipedia.org/wiki/Red%E2%80%93black_tree), presumably chosen because it offers some worst-case guarantees. It is first sorted by hash code. If the hash codes are the same, it uses the compareTo method of Comparable if the objects implement that interface, else the identity hash code.

If entries are removed from the map, the number of entries in the bucket might reduce such that this tree structure is no longer necessary. That's what the UNTREEIFY\_THRESHOLD = 6 is for. If the number of elements in a bucket drops below six, we might as well go back to using a linked list.

**More about HashMap Concept**

"**What will happen if two different objects have same hashcode**?”

Since hashcode () is same, bucket location would be same and collision occurs in hashMap, Since HashMap use a linked list to store in bucket, value object will be stored in next node of linked list.

"**How will you retrieve if two different objects have same hashcode**?”

finding bucket location , we will call keys.equals() method to identify correct node in linked list and return associated value object for that key in Java HashMap.

**"What happens On HashMap in Java if the size of the Hashmap exceeds a given threshold defined by load factor ?"**

If the size of the map exceeds a given threshold defined by load-factor e.g. if load factor is .75 it will act to re-size the map once it filled 75%. Java Hashmap does that by creating another new bucket array of size twice of previous size of hashmap, and then start putting every old element into that new bucket array and this process is called rehashing because it also applies hash function to find new bucket location.

**hashcode method is used at the time of put and get**, **equals method is used at the time of get for collision and while inserting to check for duplicate**.

**Rehashing** is a process that occurs automatically by HashMap when the number of keys in the map reaches the threshold value. The threshold value is calculated as **threshold** = capacity \* (load factor of 0.75).

Load Factor: 0.75

Initial Capacity: 16 (Available Capacity initially)

Threshold = Load Factor \* Available Capacity = 0.75 \* 16 = 12

## **What Happens If Two Keys Have the Same Hashcode?**

The same collision resolution mechanism will be used here. **key.equals(k)**will check until it is true, and if it is true, it returns the value of it.

**putIfAbsent() in ConcurrentHashMap**

A ConcurrentHashMap is designed so that it can be used by a large number of concurrent Threads.

Now, if you used the methods provided by the standard Map interface you would probably write something like this

if(!map.containsKey("something")) {

map.put("something", "a value");

}

This looks good and seems to do the job but, **it is not thread safe**. So you would then think, "Ah, but I know about the synchronized keyword" and change it to this

synchronized(map) {

if(!map.containsKey("something")) {

map.put("something", "a value");

}

}

Which fixes the issue.

Now what you have done is *locked the entire map* for *both read and write* while you check if the key exists and then add it to the map.

This is a very crude solution. Now you could implement your own solution with double checked locks and re-locking on the key etc. but that is *a lot* of *very complicated* code that is very prone to bugs.

So, instead you use the solution provided by the JDK.

The ConcurrentHashMap is a clever implementation that divides the Map into regions and locks them individually so that you can have concurrent, thread safe, reads and writes of the map without external locking.

**Like all other methods in the implementation putIfAbsent locks the key's region** **and not the whole Map** and therefore allows other things to go on in other regions in the meantime.