How does a HashMap work in JAVA

Most JAVA developers are using Maps and especially HashMaps. A HashMap is a simple yet powerful way to store and get data. But how many developers know how a HashMap works internally? A few days ago, I've read a huge part of the source code of java.util.HashMap (in Java 7 then Java 8) in order to have a deep understanding of this fundamental data structure. In this post, I'll explain the implementation of java.util.HashMap, present what's new in the JAVA 8 implementation and talk about performance, memory and known issues when using HashMaps.

Internal storage

The JAVA HashMap class implements the interface Map<K,V>. The main methods of this interface are:

- V put(K key, V value)
- V get(Object key)
- V remove(Object key)
- Boolean containsKey(Object key)

HashMaps use an inner class to store data: the Entry<K, V>. This entry is a simple key-value pair with two extra data:

- a reference to another Entry so that a HashMap can store entries like singly linked lists
- a hash value that represents the hash value of the key. This hash value is stored to avoid the computation of the hash every time the HashMap needs it.

Here is a part of the Entry implementation in JAVA 7:

A HashMap stores data into multiple singly linked lists of entries (also called **buckets** or **bins**). All the lists are registered in an array of Entry (Entry<K,V>[] array) and the default capacity of this inner array is 16.

 The following picture shows the inner storage of a HashMap instance with an array of nullable entries. Each Entry can link to another Entry to form a linked list.

All the keys with the same hash value are put in the same linked list (bucket). Keys with different hash values can end-up in the same bucket.

When a user calls put(K key, V value) or get(Object key), the function computes the index of the bucket in which the Entry should be. Then, the function iterates through the list to look for the Entry that has the same key (using the equals() function of the key).

In the case of the get(), the function returns the value associated with the entry (if the entry exists).

In the case of the put(K key, V value), if the entry exists the function replaces it with the new value otherwise it creates a new entry (from the key and value in arguments) at the head of the singly linked list.

This index of the bucket (linked list) is generated in 3 steps by the map:

- It first gets the **hashcode** of the key.
- It **rehashes** the hashcode to prevent against a bad hashing function from the key that would put all data in the same index (bucket) of the inner array
- It takes the rehashed hash hashcode and **bit-masks** it with the length (minus 1) of the array. This operation assures that the index can’t be greater than the size of the array. You can see it as a very computationally optimized modulo function.

Here is the JAVA 7 and 8 source code that deals with the index:

In order to work efficiently, the size of the inner array needs to be a power of 2, let's see why.

Imagine the array size is 17, the mask value is going to be 16 (size -1). The binary representation of 16 is 0…010000, so for any hash value H the index generated with the bitwise formula "H AND 16" is going to be either 16 or 0. This means that the array of size 17 will only be used for 2 buckets: the one at index 0 and the one at index 16, not very efficient…

But, if you now take a size that is a power of 2 like 16, the bitwise index formula is "H AND 15". The binary representation of 15 is 0…001111 so the index formula can output values from 0 to 15 and the array of size 16 is fully used. For example:

- if H = 952 , its binary representation is 0..0111011**1000**, the associated index is 0…0**1000** = 8
- if H = 1576 its binary representation is 0..01100010**1000**, the associated index is 0…0**1000** = 8
- if H = 12356146, its binary representation is 0..0101111001000101000110**0010**, the associated index is 0…0**0010** = 2
- if H = 59843, its binary representation is 0..0111010011100**0011**, the associated index is 0…0**0011** = 3

This is why the array size is a power of two. This mechanism is transparent for the developer: if he chooses a HashMap with a size of 37, the Map will automatically choose the next power of 2 after 37 (64) for the size of its inner array.

Auto resizing

After getting the index, the function (get, put or remove) visits/iterates the associated linked list to see if there is an existing Entry for the given key. Without modification, this mechanism could lead to performance issues because the function needs to iterate through the entire list to see if the entry exists. Imagine

that the size of the inner array is the default value (16) and you need to store 2 millions values. In the best case scenario, each linked list will have a size of 125 000 entries (2/16 millions). So, each get(), remove() and put() will lead to 125 000 iterations/operations. To avoid this case, the HashMap has the ability to increase its inner array in order to keep very short linked lists.

When you create a HashMap, you can specify an initial size and a loadFactor with the following constructor:

If you don't specify arguments, the default initialCapacity is 16 and the default loadFactor is 0.75. The initialCapacity represents to the size of the inner array of linked lists.

Each time you add a new key/value in your Map with put(…), the function checks if it needs to increase the capacity of the inner array. In order to do that, the map stores 2 data:

- The size of the map: it represents the number of entries in the HashMap. This value is updated each time an Entry is added or removed.
- A threshold: it's equal to (capacity of the inner array) * loadFactor and it is refreshed after each resize of the inner array

Before adding the new Entry, put(…) checks if size > threshold and if it the case it recreates a new array with a doubled size. Since the size of the new array has changed, the indexing function (which returns the bitwise operation "hash(key) AND (sizeOfArray-1)") changes. So, the resizing of the array creates twice more buckets (i.e. linked lists) and **redistributes all the existing entries** into the buckets (the old ones and the newly created).

This aim of this resize operation is to decrease the size of the linked lists so that the time cost of put(), remove() and get() methods stays low. All entries whose keys have the same hash will stay in the same bucket after the resizing. But, 2 entries with different hash keys that were in the same bucket before might not be in the same bucket after the transformation.

The picture shows a representation before and after the resizing of the inner array. Before the increase, in order to get Entry E, the map had to iterate through

a list of 5 elements. After the resizing, the same get() just iterates through a linked list of 2 elements, the get() is 2 times faster after the resizing !

Note: the HashMap only increases the size of the inner array, it doesn't provide a way to decrease it.

Thread Safety

If you already know HashMaps, you know that is not threads safe, but why? For example imagine that you have a Writer thread that puts only new data into the Map and a Reader thread that reads data from the Map, why shouldn't it work?

Because during the auto-resizing mechanism, if a thread tries to put or get an object, the map might use the old index value and won't find the new bucket in which the entry is.

The worst case scenario is when 2 threads put a data at the same time and the 2 put() calls resize the Map at the same time. Since both threads modify the linked lists at the same time, the Map might end up with an inner-loop in one of its linked lists. If you tries to get a data in the list with an inner loop, the get() will never end.

The **HashTable** implementation is a thread safe implementation that prevents from this situation. But, since all the CRUD methods are synchronized this implementation is very slow. For example, if thread 1 calls get(key1), thread 2 calls get(key2) and thread 3 calls get(key3), only one thread at a time will be able to get its value whereas the 3 of them could access the data at the same time.

A smarter implementation of a thread safe HashMap exists since JAVA 5: the **ConcurrentHashMap**. Only the buckets are synchronized so multiples threads can get(), remove() or put() data at the same time if it doesn't imply accessing the same bucket or resizing the inner array. **It's better to use this implementation in a multithreaded application**.

Key immutability

Why Strings and Integers are a good implementation of keys for HashMap? Mostly because they are **immutable**! If you choose to create your own Key class and don't make it immutable, you might lose data inside the HashMap.

Look at the following use case:

- You have a key that has an inner value "1"
- You put an object in the HashMap with this key
- The HashMap generates a hash from the hashcode of the Key (so from "1")
- The Map **stores this hash** in the newly created Entry
- You modify the inner value of the key to "2"
- The hash value of the key is modified but the HashMap doesn't know it (because the old hash value is stored)
- You try to get your object with your modified key
- The map computes the new hash of your key (so from "2") to find in which linked list (bucket) the entry is
    - Case 1: Since you modified your key, the map tries to find the entry in the wrong bucket and doesn't find it
    - Case 2: Luckily, the modified key generates the same bucket as the old key. The map then iterates through the linked list to find the entry with the same key. But to find the key, **the map first compares the hash values** and then calls the equals() comparison. Since your modified key doesn't have the same hash as the old hash value (stored in the entry), the map won't find the entry in the linked-list.

Here is a concrete example in Java. I put 2 key-value pairs in my Map, I modify the first key and then try to get the 2 values. Only the second value is returned from the map, the first value is "lost" in the HashMap:

The output is: "test1= null test2=test 2". As expected, the Map wasn't able to retrieve the string 1 with the modified key 1.

JAVA 8 improvements

The inner representation of the HashMap has changed a lot in JAVA 8. Indeed, the implementation in JAVA 7 takes 1k lines of code whereas the implementation in JAVA 8 takes 2k lines. Most of what I've said previously is true except the linked lists of entries. In JAVA8, you still have an array but it now stores Nodes that contains the exact same information as Entries and therefore are also linked lists:

Here is a part of the Node implementation in JAVA 8:

So what's the big difference with JAVA 7? Well, Nodes can be extended to TreeNodes. A TreeNode is a red-black tree structure that stores really more information so that it can add, delete or get an element in O(log(n)).

FYI, here is the exhaustive list of the data stored inside a TreeNode

Red black trees are self-balancing binary search trees. Their inner mechanisms ensure that their length is always in log(n) despite new adds or removes of nodes. The main advantage to use those trees is in a case where many data are in the same index (bucket) of the inner table, the search in a tree will cost **O(log(n))** whereas it would have cost **O(n)** with a linked list.

As you see, the tree takes really more space than the linked list (we'll speak about it in the next part).

By inheritance, the **inner table can contain** both Node (**linked list**) **and** TreeNode (red-black **tree**). Oracle decided to use both data structures with the following rules:
– If for a given index (bucket) in the inner table there are more than 8 nodes, the linked list is transformed into a red black tree
– If for a given index (bucket) in the inner table there are less than 6 nodes, the tree is transformed into a linked list

This picture shows an inner array of a JAVA 8 HashMap with both trees (at bucket 0) and linked lists (at bucket 1,2 and 3). Bucket 0 is a Tree because it has more than 8 nodes.

Memory overhead

JAVA 7

The use of a HashMap comes at a cost in terms of memory. In JAVA 7, a HashMap wraps key-value pairs in Entries. An entry has:

- a reference to a next entry
- a precomputed hash (integer)
- a reference to the key
- a reference to the value

Moreover, a JAVA 7 HashMap uses an inner array of Entry. Assuming a JAVA 7 HashMap contains N elements and its inner array has a capacity CAPACITY, the extra memory cost is approximately:

sizeOf(integer)* N + sizeOf(reference)* (3*N+C)

Where:

- the size of an integer depends equals 4 bytes
- the size of a reference depends on the JVM/OS/Processor but is often 4 bytes.

Which means that the overhead is often 16 * N + 4 * CAPACITY bytes

Reminder: after an auto-resizing of the Map, the CAPACITY of the inner array equals the next power of two after N.

Note: Since JAVA 7, the HashMap class has a lazy init. That means that even if you allocate a HashMap, the inner array of entry (that costs 4 * CAPACITY bytes) won't be allocated in memory until the first use of the put() method.

JAVA 8

With the JAVA 8 implementation, it becomes a little bit complicated to get the memory usage because a Node can contain the same data as an Entry or the same data plus 6 references and a Boolean (if it's a TreeNode).

If the all the nodes are only Nodes, the memory consumption of the JAVA 8 HashMap is the same as the JAVA 7 HashMap.

If the all the nodes are TreeNodes, the memory consumption of a JAVA 8 HashMap becomes:

N * sizeOf(integer) + N * sizeOf(boolean) + sizeOf(reference)* (9*N+CAPACITY)

In most standards JVM, it's equal to 44 * N + 4 * CAPACITY bytes

Performance issues

Skewed HashMap vs well balanced HashMap

In the best case scenario, the get() and put() methods have a O(1) cost in time complexity. But, if you don't take care of the hash function of the key, you might end up with very slow put() and get() calls. The good performance of the put() and get depends on the repartition of the data into the different indexes of the inner array (the buckets). If the hash function of your key is ill-designed, you'll have a skew repartition (no matter how big the capacity of the inner array is). All the put() and get() that use the biggest linked lists of entry will be slow because they'll need to iterate the entire lists. In the worst case scenario (if most of the data are in the same buckets), you could end up with a O(n) time complexity.
Here is a visual example. The first picture shows a skewed HashMap and the second picture a well balanced one.

In the case of this skewed HashMap the get()/put() operations on the bucket 0 are costly. Getting the Entry K will cost 6 iterations

In the case of this well balanced HashMap, getting the Entry K will cost 3 iterations. Both HashMaps store the same amount of data and have the same inner array size. The only difference is the hash (of the key) function that distributes the entries in the buckets.

Here is an extreme example in JAVA where I create a hash function that puts all the data in the same bucket then I add 2 million elements.

On my core i5-2500k @ 3.6Ghz it takes **more than 45 minutes** with java 8u40 (I stopped the process after 45 minutes).

Now, If I run the same code but this time I use the following hash function

it takes **46 seconds**, which is way better! This hash function has a better repartition than the previous one so the put() calls are faster.

And If I run the same code with the following hash function that provides an even better hash repartition

it now takes **2 seconds**.

I hope you realize how important the hash function is. If a ran the same test on JAVA 7, the results would have been worse for the first and second cases (since the time complexity of put is O(n) in JAVA 7 vs O(log(n)) in JAVA 8)

When using a HashMap, you need to find a hash function for your keys that **spreads the keys into the most possible buckets**. To do so, you need to **avoid hash collisions**. The String Object is a good key because of it has good hash function. Integers are also good because their hashcode is their own value.

Resizing overhead

If you need to store a lot of data, you should create your HashMap with an initial capacity close to your expected volume.

If you don’t do that, the Map will take the default size of 16 with a factorLoad of 0.75. The 11 first put() will be very fast but the 12th (16*0.75) will recreate a new inner array (with its associated linked lists/trees) with a new capacity of 32. The 13th to 23th will be fast but the 24th (32*0.75) will recreate (again) a costly new representation that doubles the size of the inner array. The internal resizing operation will appear at the 48th, 96th,192th, … call of put(). At low volume the full recreation of the inner array is fast but at high volume it can takes seconds to minutes. By initially setting your expected size, you can **avoid** these **costly operations**.

But there is a **drawback**: if you set a very high array size like 2^28 whereas you’re only using 2^26 buckets in your array, you will **waste** a lot of **memory** (approximately 2^30 bytes in this case).

Conclusion

For simple use cases, you don’t need to know how HashMaps work since you won’t see the difference between a O(1) and a O(n) or O(log(n)) operation. But it’s always better to understand the underlaying mecanism of one of the most used data structures. Moreover, for a java developer position it’s a typical interview question.

At high volume it becomes important to know how it works and to understand the importance of the hash function of the key.

I hope this article helped you to have a deep understanding of the HashMap implementation.