HashMap Working

--------------------------

HashMap in Java works on hashing principle. It is a data structure which allows us to store object and retrieve it in constant time O(1) provided we know the key. In hashing, hash functions are used to link key and value in HashMap. Objects are stored by calling put(key, value) method of HashMap and retrieved by calling get(key) method. When we call put method, hashcode() method of the key object is called so that hash function of the map can find a bucket location to store value object, which is actually an index of the internal array, known as the table. HashMap internally stores mapping in the form of **Map.Entry** object which contains both key and value object. When you want to retrieve the object, you call [the get() method](http://java67.blogspot.com/2013/06/how-get-method-of-hashmap-or-hashtable-works-internally.html) and again pass the key object. This time again key object generate same hash code (it's mandatory for it to do so to retrieve the object and that's why HashMap keys are immutable e.g. String) and we end up at same bucket location. If there is only one object then it is returned and that's your value object which you have stored earlier. Things get little [tricky](http://java67.blogspot.com/2012/09/top-10-tricky-java-interview-questions-answers.html) when collisions occur. It's easy to answer this question if you have read good book or course on data structure and algorithms like [this](https://pluralsight.pxf.io/c/1193463/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fads-part1)one. If you know how hash table data structure works then this is a piece of cake.  
  
Since the internal array of HashMap is of fixed size, and if you keep storing objects, at some point of time hash function will return same bucket location for two different keys, this is called collision in HashMap. In this case, a linked list is formed at that bucket location and a new entry is stored as next node.  
  
If we try to retrieve an object from this linked list, we need an extra check to search correct value, this is done by equals() method. Since each node contains an entry, HashMap keeps comparing entry's key object with the passed key using equals() and when it return true, Map returns the corresponding value.  
  
  
Since searching inlined list is O(n) operation, in worst case hash collision reduce a map to linked list. This issue is recently addressed in Java 8 by replacing linked list to the tree to search in O(logN) time. By the way, you can easily verify how HashMap works by looking at the code of HashMap.java in your Eclipse IDE if you know [how to attach source code of JDK in Eclipse](http://javarevisited.blogspot.com/2012/12/how-to-attach-source-in-eclipse-Jar-JDK-debugging.html).  
  
  
How HashMap works in Java or sometimes how does get method work in HashMap is a very common question on Java interviews nowadays. Almost everybody who worked in Java knows about HashMap, where to use HashMap and difference between Hashtable and HashMap then why this interview question becomes so special? Because of the depth it offers.  
  
It has become very [popular Java interview question](http://javarevisited.blogspot.com/2015/10/133-java-interview-questions-answers-from-last-5-years.html) in almost any senior or mid-senior level Java interviews. Investment banks mostly prefer to ask this question and sometimes even ask you to implement your own HashMap based upon your coding aptitude. The introduction of [ConcurrentHashMap](http://javarevisited.blogspot.co.uk/2013/02/concurrenthashmap-in-java-example-tutorial-working.html) and other concurrent collections has also made this questions as starting point to delve into a more advanced feature. let's start the journey.

## How HashMap Internally Works in Java

Questions start with simple statement:

**Have you used HashMap before**or**What is HashMap? Why do you use it**

Almost everybody answers this with yes and then interviewee keep talking about common facts about HashMap like HashMap accept null while Hashtable doesn't, [HashMap is not synchronized](http://javarevisited.blogspot.com/2010/10/difference-between-hashmap-and.html), HashMap is fast and so on along with basics like its stores key and value pairs etc. This shows that person has used HashMap and quite familiar with the functionality it offers, but interview takes a sharp turn from here and next set of follow-up questions gets more detailed about fundamentals involved with HashMap in Java. Interviewer strike back with questions like:

**Do you Know how HashMap works in Java** or **How does get () method of HashMap works in Java**

And then you get answers like,  I don't bother its standard Java API, you better look code on Java source or Open JDK; I can find it out in Google at any time etc. But some interviewee definitely answers this and will say **HashMap works on the principle of hashing**, we have put(key, value) and get(key) method for storing and retrieving Objects from HashMap. When we pass Key and Value object  to put() method on Java HashMap, HashMap implementation calls [hashCode method](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html)on Key object and applies returned hashcode into its own hashing function to find a bucket location for storing Entry object, important point to mention is that HashMap in Java stores both key and value object as Map.Entry in a bucket which is essential to understand the retrieving logic.   
  
If people fail to recognize this and say it only stores Value in the bucket they will fail to explain the retrieving logic of any object stored in Java HashMap. This answer is very much acceptable and does make sense that interviewee has a fair bit of knowledge on how hashing works and how HashMap  works in Java. But this is just start of story and confusion increases when you put interviewee on scenarios faced by Java developers on day by day basis. Next question could be about collision detection and collision resolution in Java HashMap  e.g.

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

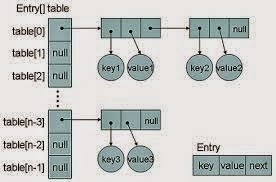
Now from here onwards real confusion starts, sometime candidate will say that since hashcode is equal, both objects are equal and HashMap  will throw exception or not store them again etc, Then you might want to remind them about [equals() and hashCode() contract](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html)that two unequal objects in Java can have same hashcode. Some will give up at this point and few will move ahead and say "Since hashcode is same, bucket location would be same and collision will occur in HashMap Since HashMap uses LinkedList to store object, this entry (object of Map.Entry comprise key and value )  will be stored in [LinkedList](http://javarevisited.blogspot.sg/2012/02/difference-between-linkedlist-vs.html). Great this answer make sense though there are many collision resolution methods available  like linear probing and chaining, this is simplest and HashMap in Java does follow this. But story does not end here and interviewer asks

**How will you retrieve Value object  if two Keys will have the same hashcode?**

Interviewee will say we will call get() method and then HashMap uses Key Object's hashcode to find out bucket location and retrieves Value object but then you need to remind him that there are two Value objects are stored in same bucket , so they will say about [traversal in LinkedList](http://javarevisited.blogspot.sg/2010/10/how-do-you-find-length-of-singly-linked.html)until we find the value object , then you ask how do you identify value object because you don't  have value object to compare ,Until they know that HashMap  stores both Key and Value in LinkedList node or as Map.Entry they won't be able to resolve this issue and will try and fail.

But those bunch of people who remember this key information will say that after finding bucket location, we will **call keys.equals() method** to identify a correct node in LinkedList and return associated value object for that key in Java HashMap. Perfect this is the correct answer.

In many cases interviewee fails at this stage because they get confused between[hashCode()](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html) and equals(**)** or keys and values object in Java HashMap  which is pretty obvious because they are dealing with the hashcode() in all previous questions and equals() come in picture only in case of retrieving value object from HashMap in Java. Some good developer point out here that using immutable, [final object](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) with proper equals() and hashcode() implementation would act as perfect Java HashMap  keys and**improve the performance of Java HashMap  by reducing collision**. Immutability also allows caching their hashcode of different keys which makes overall retrieval process very fast and suggest that [String](http://javarevisited.blogspot.sg/2011/07/string-vs-stringbuffer-vs-stringbuilder.html)and various wrapper classes e.g. Integer very good keys in Java HashMap.

[](https://4.bp.blogspot.com/-adRczhctozE/VD_eimhTQbI/AAAAAAAACCg/lfA1G5GZXyM/s1600/How+HashMap+works+in+Java+(1).jpg)

Now if you clear this entire Java HashMap interview,  You will be surprised by this very interesting question "**What happens On HashMap in Java if the size of the HashMap  exceeds a given threshold defined by load factor ?"**. Until you know how HashMap  works exactly you won't be able to answer this question. If the size of the Map exceeds a given threshold defined by load-factor e.g. if the load factor is .75 it will act to re-size the map once it filled 75%. Similar to other collection classes like [ArrayList](http://javarevisited.blogspot.sg/2011/05/example-of-arraylist-in-java-tutorial.html),  Java HashMap re-size itself by creating a new bucket array of size twice of the previous size of HashMap and then start putting every old element into that new bucket array. This process is called rehashing because it also applies the hash function to find new bucket location.

If you manage to answer this question on HashMap in Java you will be greeted by **"do you see any problem with resizing of HashMap  in Java"**, you might not be able to pick the context and then he will try to give you hint about multiple thread accessing the Java HashMap and potentially looking for **race condition on HashMap  in Java**.

So the answer is Yes there is potential [race condition](http://javarevisited.blogspot.sg/2012/02/what-is-race-condition-in.html) exists while resizing HashMap in Java, if two [thread](http://javarevisited.blogspot.sg/2011/02/how-to-implement-thread-in-java.html)at the same time found that now HashMap needs resizing and they both try to resizing. on the process of resizing of HashMap in Java, the element in the bucket which is stored in linked list get reversed in order during their migration to new bucket because Java HashMap  doesn't append the new element at tail instead it append new element at the head to avoid tail traversing. If race condition happens then you will end up with an infinite loop. Though this point, you can potentially argue that what the hell makes you think to use HashMap  in multi-threaded environment to interviewer :)

## Some more Hashtable and HashMap Questions

Few more question on HashMap in Java which is contributed by readers of Javarevisited blog:

**1) Why String, Integer and other wrapper classes are considered good keys?**

String, Integer and other wrapper classes are natural candidates of HashMap key, and String is most frequently used key as well because [String is immutable and final](http://javarevisited.blogspot.sg/2010/10/why-string-is-immutable-in-java.html), and overrides equals and hashcode() method. Other wrapper class also shares similar property. Immutability is required, in order to prevent changes on fields used to calculate hashCode() because if key object returns different hashCode during insertion and retrieval than it won't be possible to get an object from HashMap.   
  
Immutability is best as it offers other advantages as well like [thread-safety](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html), If you can  keep your hashCode same by only making certain fields final, then you go for that as well. Since equals() and hashCode() method is used during retrieval of value object from HashMap, it's important that key object correctly override these methods and follow contact. If unequal object returns different hashcode than chances of collision will be less which subsequently improve the performance of HashMap.

**2) Can we use any custom object as a key in HashMap?**

This is an extension of previous questions. Of course you can use any Object as key in Java HashMap provided it follows equals and hashCode contract and its hashCode should not vary once the object is inserted into [Map](http://javarevisited.blogspot.sg/2011/12/how-to-traverse-or-loop-hashmap-in-java.html). If the custom object is Immutable than this will be already taken care because you can not change it once created.

**3) Can we use ConcurrentHashMap in place of Hashtable?**

This is another question which getting popular due to increasing popularity of ConcurrentHashMap. Since we know Hashtable is synchronized but ConcurrentHashMap provides better concurrency by only locking portion of map determined by concurrency level. ConcurrentHashMap is certainly introduced as Hashtable and can be used in place of it, but Hashtable provides stronger thread-safety than ConcurrentHashMap. See my post [difference between Hashtable and ConcurrentHashMap](http://javarevisited.blogspot.sg/2011/04/difference-between-concurrenthashmap.html) for more details.

Personally, I like this question because of its depth and number of concept it touches indirectly if you look at questions asked during interview this HashMap  questions has verified

* The concept of hashing
* Collision resolution in HashMap
* Use of equals () and hashCode () and their importance in HashMap?
* The benefit of the immutable object?
* Race condition on HashMap  in Java
* Resizing of Java HashMap

Just to summarize here are the answers which do make sense for above questions

**How HashMap  works in Java**

HashMap  works on the principle of hashing, we have put() and get() method for storing and retrieving object from HashMap.When we pass both key and value to put() method to store on HashMap, it uses key object hashcode() method to calculate hashcode and them by applying hashing on that hashcode it identifies bucket location for storing value object. While retrieving it uses key object equals method to find out correct key value pair and return value object associated with that key. HashMap  uses linked list in case of collision and object will be stored in next node of linked list. Also, [HashMap stores both key and value tuple](http://java67.blogspot.com/2013/02/10-examples-of-hashmap-in-java-programming-tutorial.html) in every node of linked list in the form of Map.Entry object.

**What will happen if two different HashMap  key objects have the same hashcode?**

They will be stored in the same bucket but no next node of linked list. And keys equals () method will be used to identify correct key value pair in HashMap.

**How null key is handled in HashMap? Since equals() and hashCode() are used to store and retrieve values, how does it work in case of the null key?**  
The null key is handled specially in HashMap, there are two separate methods for that putForNullKey(V value) and getForNullKey(). Later is offloaded version of get() to look up null keys.  Null keys always map to index 0.  This null case is split out into separate methods for the sake of performance in the two most commonly used operations (get and put), but incorporated with conditionals in others. In short, equals() and hashcode() method are not used in case of null keys in HashMap.  
  
here is how nulls are retrieved from HashMap  
  
   **private** V **getForNullKey**() {

**if** (size == **0**) {

**return** **null**;

}

**for** (Entry<K,V> e = table[**0**]; e != **null**; e = e.next) {

**if** (e.key == **null**)

**return** e.value;

}

**return** **null**;

}

In terms of usage, Java HashMap is very versatile and I have mostly used HashMap as cache in an electronic trading application I have worked. Since finance domain used Java heavily and due to performance reason we need caching HashMap and ConcurrentHashMap  comes as very handy there. You can also check following articles from Javarevisited to learn more about HashMap and Hashtable in Java:

## HashMap Changes in JDK 1.7 and JDK 1.8

There is some [performance improvement done on HashMap and ArrayList from JDK 1.7](http://javarevisited.blogspot.com/2014/07/java-optimization-empty-arraylist-and-Hashmap-cost-less-memory-jdk-17040-update.html), which reduce memory consumption. Due to this empty Map are lazily initialized and will cost you less memory. Earlier, when you create HashMap e.g. new HashMap() it automatically creates an array of default length e.g. 16. After some research, Java team found that most of this Map are temporary and never use that many elements, and only end up wasting memory. Also, From JDK 1.8 onwards HashMap has introduced an improved strategy to deal with high collision rate. Since a poor hash function e.g. which always return location of same bucket, can turn a HashMap into linked list, i.e. converting get() method to perform in O(n) instead of O(1) and someone can take advantage of this fact, Java now internally replace linked list to a binary true once certain threshold is breached. This ensures performance or order O(log(n)) even in the worst case where a hash function is not distributing keys properly.

# [**What is Load factor and Rehashing in Hashmap?**](http://javabypatel.blogspot.com/2015/10/what-is-load-factor-and-rehashing-in-hashmap.html)

IN [JAVA](http://javabypatel.blogspot.com/search/label/Java) - ON 03:11:00 - [NO COMMENTS](http://javabypatel.blogspot.com/2015/10/what-is-load-factor-and-rehashing-in-hashmap.html#comment-form)

### What is Load factor and Rehashing in Hashmap?

**This is the famous interview question for experienced, So Let's see what it is all about.**   
  
Hashmap is very popular data structure and found useful for solving many problems due to O(1) time complexity for both get and put operation.

Before understanding Load Factor and Rehashing, It is important to understand below articles,   
So please go through it if you are not awareof**,**  
  
[**What is Hashmap & How hashmap API works?**](http://javabypatel.blogspot.in/2015/09/hashmap-data-structure-and-hashcode.html)  
[**What is Hashcode and How hashmap uses it?**](http://javabypatel.blogspot.in/2015/10/how-hashmap-works-internally-explain-relation-with-hashcode-and-equals-method.html)  
[**How time complexity of Hashmap Put and Get operation is O(1)?**](http://javabypatel.blogspot.in/2015/10/time-complexity-of-hashmap-get-and-put-operation.html)

### Load Factor

**When the total number of items in hashmap goes on increasing keeping the default initial capacity of hashmap 16, At one point of time, hashmap performance will start degrading and need to increase buckets for improving performance.**  
  
**Load Factor is a measure, which decides when exactly to increase the hashmap capacity(buckets) to maintain get and put operation complexity of O(1).**

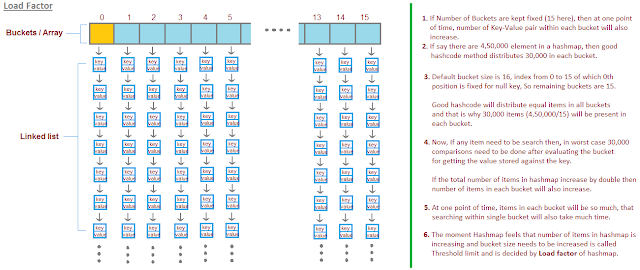
**Default load factor of Hashmap is 0.75f (i.e 75% of current map size).**

You can also say, load factor is a measure **"Till what load, hashmap can allow elements to put in it before its capacity is automatically increased"**  
  
Above line will make more sense with the help of an example,

**Default capacity of Hashmap is 2^4 = 16 buckets.**

Let say we have well implemented hashcode() method, which make sure that key-value pair will be well distributed across 16 buckets equally.  
  
So, If there are 16 items in hashmap, then good hashcode method will distribute 1 item in each bucket. **Searching for any item in this case will take only 1 look up.**  
  
Now, If there are 32 items in hashmap, then good hashcode method will distribute 2 item in each bucket.**Searching for any item in this case will take maximum 2 look up.**  
  
Now, If there are 128 items in hashmap, then good hashcode method will distribute 8 item in each bucket.**Searching for any item in this case will take maximum 8 look up.**  
  
**If you observe, If the number of items in hashmap is doubled, still the maximum look up time in each bucket is not increasing very high and remain almost constant.**

**If say, the number of items goes on increasing in the map, then what will happen?**  
If the amount of item keeps on increasing and the number of buckets are fixed(16) then at one time, performance of hashmap will start degrading due to large number of items in each bucket. 

**[](http://2.bp.blogspot.com/-vSkLhOhPuwo/VhFqKx0-GNI/AAAAAAAAAdI/DnZ5gkiU_Po/s1600/hashmapLoadFactor.png)**

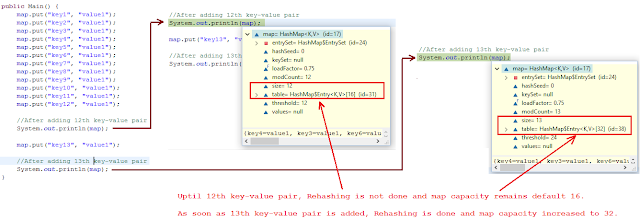
Now, say if there are 5,00,000 items in hashmap, then, good hashcode method will distribute 31,250 items in each bucket. **Searching for any item in this case, will require maximum 31,250 look up.**  
  
Compare to total number of items in hashmap, look up required for searching any item within bucket is very less, but still expensive, as now there are 31,250 items present in each bucket.   
So in worst case it has to compare 31,250 items for both put and get operation.  
  
**Just double the total items from 5,00,000 to 10,00,000, each bucket now will have 62,500 items and this time searching a item will really hit performance.**

**So what is the solution for this????**  
  
Initially we was doing good, when bucket size was more(16) and total items was less. When total items keep growing, at one point, our performance start degrading due to much items present in each bucket. So what do you think problem is???  
  
**Problem is, keeping bucket size fixed(16), we kept on increasing the total number of items in map and that disturbed time complexity.  
  
If we increase the total number of buckets, when total items in each bucket starts increasing, then we will be able to keep constant number of items in each bucket and maintain the time complexity of O(1) for get and put operation.**

**The decision of "When to increase the number of buckets" is decided by Load Factor.   
  
Load Factor is a measure which decides when exactly to increase the hashmap capacity or you can say bucket capacity, so that get and put operation can still have O(1) complexity.**   
  
**Default, initial capacity of the HashMap is 16 and Load factor is 0.75**

**So, when to increase the hashmap size is decided by product of,**  
**(initial capacity of hashmap \* Load factor of hashmap).**

**Lets see, when initial size of hashmap will be increased based on above forumla,**  
**initial capacity of hashmap \* Load factor of hashmap =  16 \* 0.75 = 12.**  
 **This represents that uptil 12th key-value pair hashmap will keep its size to 16 and as soon as 13th item(key-value pair) will come into the Hashmap,  it will increase its size from default 2^4 = 16 buckets to 2^5 = 32 buckets.**

**[](https://3.bp.blogspot.com/-MGAW9KpCIIg/WLJYNNnvU9I/AAAAAAAABrg/cmswnZYYqAsPFz3AZjLDfsYxTCzqLuo5QCEw/s1600/loadfactor-rehash-in-hashmap.png)**

**Hashmap grows in power of 2^n. Initially n=4 and then keep on growing when threshold limit is reached.**

**Another way of calculating, when hashmap size will be doubled.**

**When Load factor ratio (m/n) reaches 0.75 at that time hashmap increases its capacity, where n = the total size of the hash map and m = number of entries in a map.**

Lets take a example,

Default bucket size size if 16. First element came in, do we need to increase the hashmap capacity is decided by,  **size of hashmap / number of buckets = 1/16 =  0.0625.**

**Compare, 0.0625 > 0.75 Load factor ? No. So no need to increase the map size.**

11th element came in, do we need to increase the hashmap capacity, 11/16 =  0.6875

**Compare 0.6875 > 0.75 Load factor ? No. So no need to increase the map size.**

12th element came in, do we need to increase the hashmap capacity, 12/16 =  0.75

**Compare 0.75 > 0.75 Load factor ? No. So no need to increase the map size.**

13th element came in, do we need to increase the hashmap capacity, 13/16 =  0.81

**Compare 0.81 > 0.75 Load factor ? Yes. We need to increase the map size now**.

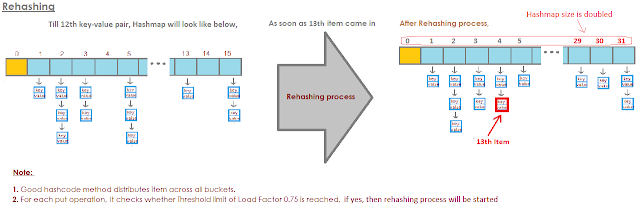
**It is advisable to have a load factor of around 0.75 for keeping the put and get complexity around O(1).**

**NOTE:**   
Load Factor and Initial Capacity(Number of buckets) can be configured while creation of Hashmap like shown below,

**HashMap m = new HashMap(int initialCapacity, float loadFactor);**

### Rehashing

**Rehashing is the process of re-calculating the hashcode of already stored entries (Key-Value pairs), to move them to another bigger size hashmap when Load factor threshold is reached.**  
  
When the number of items in map, crosses the Load factor limit at that time hashmap doubles its capacity and hashcode is re-calculated of already stored elements for even distribution of key-value pairs across new buckets.  
  
**Why Rehashing is required?**  
  
**After doubling the capacity, what to do with the key-value pairs already present in buckets?**  
If we keep the existing key-value pairs as it is, then doubling the capacity may not help,   
because O(1) complexity will be achieved only if items are evenly distributed across all buckets.  
  
**So for each existing key-value pairs, hashcode is calculated again with increased hashmap capacity as a parameter, which results in either placing the item in same bucket or in different bucket.**

[](https://3.bp.blogspot.com/-NE2005vXASU/WS8ILuLUobI/AAAAAAAAB4o/QxWu10vy9TkoS8_nkrm7CAwDML3WP6S1wCEw/s1600/hashmap-rehashing-process.png)

**When the size of hashmap is changed, the process of re-calculating the hashcode of already placed key-value pair again is known as Rehashing.**

**Rehashing is done to distribute items across the new length hashmap, so that get and put operation time complexity remains O(1).  
  
NOTE:**

**Hashmap maintain complexity of O(1) while inserting data in and getting data from hashmap, but for 13th key-value pair, put request will no longer be O(1), because as soon as map will realize that 13th element came in, that is 75% of map is filled.   
  
It will first double the bucket(array) capacity and then it will go for Rehash.**  
**Rehashing requires re-computing hashcode of already placed 12 key-value pairs again and putting them at new index which requires time.**

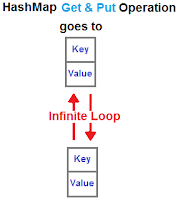
**But overall time complexity provided by hashmap, which is O(1) for get and put operations, will amortize Rehashing process over long run.**

# [**Infinite loop in HashMap**](http://javabypatel.blogspot.com/2016/01/infinite-loop-in-hashmap.html)

IN [INTERVIEWS](http://javabypatel.blogspot.com/search/label/Interviews), [JAVA](http://javabypatel.blogspot.com/search/label/Java) - ON 09:17:00 - [NO COMMENTS](http://javabypatel.blogspot.com/2016/01/infinite-loop-in-hashmap.html#comment-form)

### Why HashMap should not be used in multi threaded environment?  Can it cause infinite loop as well? When get method go to infinite loop in HashMap?

**If HashMap is used in Multi threading environment, there are chances that Get and Put operation can lead you to Infinite loop.**

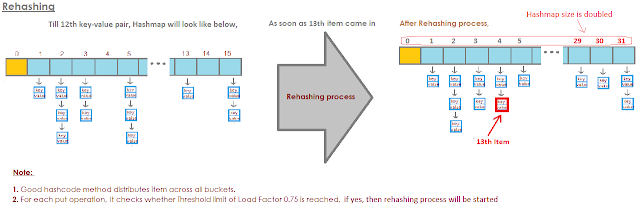
[](http://3.bp.blogspot.com/-tHmmj6H5JHY/Von-HEOBlAI/AAAAAAAAA3k/YQXgCmSeq0g/s1600/hashmap-get-operation-goes-to-infinite-loop.png)

### Overview

Before going into details on how Hashmap get/put operation can lead you to infinte loop,  Let's understand Rehashing.

### What is Rehashing?

Default capacity of HashMap is 16 and Load factor is 0.75, which means HashMap will double its capacity when 13thKey-Value pair enters in map (**16 \* 0.75 = 12**).  
  
Till 12th Key-value pair, Hashmap will keep putting items in map and as soon as you try to put 13th key-value pair, rehashing process starts.   
  
**Load factor:** Load factor is a measure "Till what load, hashmap can allow elements to put in it before its size is increased."  
  
**For more details on Load Factor and Rehashing, Please refer,**[**What is Load factor and Rehashing in Hashmap?**](http://javabypatel.blogspot.in/2015/10/what-is-load-factor-and-rehashing-in-hashmap.html)

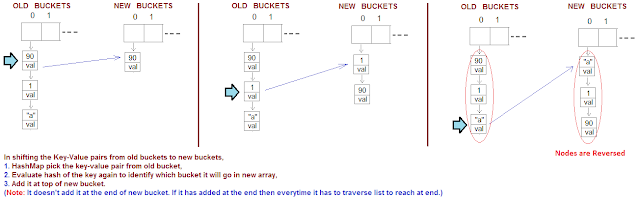
[](https://3.bp.blogspot.com/-NE2005vXASU/WS8ILuLUobI/AAAAAAAAB4k/sI0Efo5febMXsQ-vvNG46_QlCnj0YvkzQCLcB/s1600/hashmap-rehashing-process.png)

### Rehasing reverses ordering of the nodes

**In Rehashing process,**

1. Hashmap creates a New Array(Buckets) of double size first.
2. Hashmap transfers key-value pairs from Old buckets to New buckets.
3. Key-value pairs will be reversed in New buckets because Hashmap will add key-value pairs at the start in the New bucket and not at the end.
4. Hashmap adds new key-value pairs at start to avoid traversing linked list every time and keep constant performance.

**Let's see how Transferring process works with example,**

[](https://2.bp.blogspot.com/--468RN0yVwo/WS-aeVsyvGI/AAAAAAAAB48/K-0uxKsKGq8FLOpGr1Vc4tHOkeJHzYdawCLcB/s1600/hashmap-infinite-loop-working.png)

### What will happen when 2 Threads try to put 13th Key-Value pair in Hashmap?

When 2 Thread tries to access HashMap simultaneously, then you may encounter infinite Loop.  
Let see how it happens,  
  
For clarity, Lets name 2 Threads as **Thread 1** and **Thread 2,** and both try to put 13th key-value pair.

It is obvious that before putting 13th key-value pair, Hashmap has to first do Rehashing process as 13th key-value pair crossed the load factor limit.

Also, Hashmap here is accessed by Thread1 and Thread2, So it is not guaranteed which Thread will get access first.   
  
We assume that both Threads reach to a place, where they both identify that Load factor limit has crossed and maps needs Rehashing. That is where both Threads will try to call below method for transferring key-value pairs from Old buckets to New buckets.

### Method transfer() is called for transferring key-value pairs from  Old buckets to New buckets

[?](http://javabypatel.blogspot.com/2016/01/infinite-loop-in-hashmap.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | void transfer(Entry[] newTable) {      Entry[] src = table;      int newCapacity = newTable.length;      for (int j = 0; j &lt; src.length; j++) {          Entry<k> e = src[j];          if (e != null) {              src[j] = null;                // --------- Below piece of code creates a problem  ---------              do {                    Entry<k> next = e.next;                  int i = indexFor(e.hash, newCapacity);                  e.next = newTable[i];                  newTable[i] = e;                  e = next;                } while (e != null);              // --------- Till Here  ---------            }      }  } |

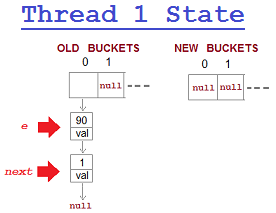
### Lets see how Hashmap ends in Infinite loop.

### Below you will see Thread 1 and Thread 2 Steps in short for quick go through.

### Note: In case if you find difficulty in understanding Thread steps, you can go to later section, where transfer() method and Thread Steps is explained in much detail.

### Thread 1 got a chance for execution.

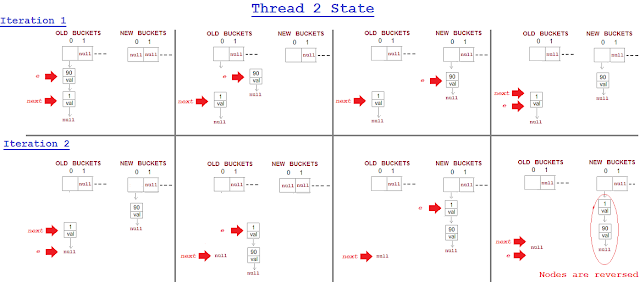
1. After executing first line inside loop(in transfer() method), Thread 1 points to first key-value pair and next(second) key-value pair to start transfer process.     
     
   **Before it execute any steps, Thread 1 loose the control and Thread 2 got chance for execution.**
2. So, Current state of Thread 1 is,  **e = Node 90**and **next = Node 1.**

[](https://2.bp.blogspot.com/-A5Na534dkIg/WTQBoSnyuvI/AAAAAAAACAY/VZuonpnnyQg_ISIPT3ZqdG5HNoRkc2xLACLcB/s1600/hashmap-infinite-loop-thread-1-steps-summary-before.png)

### Thread 2 got a chance for execution.

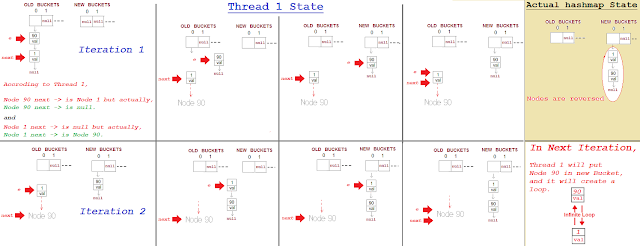
1. Fortunately, Thread 2 executed complete transfer() method without loosing control to other Thread.
2. While transferring key-value pairs from old buckets to new buckets, key-value pairs will be reversed in New buckets because hashmap will add key-value pairs at the start and not at the end. It does this to avoid traversing linked list every time and keep constant performance.
3. **Thread 2 will transfer all key-value pairs from Old buckets to New buckets and Thread 1 will get chance for execution.**

**After Thread 2 execution, Hashmap state will be as shown below,**

[](https://1.bp.blogspot.com/-ugeNUM6Mj5M/WTQC1HhHnII/AAAAAAAACAk/ZUpu6hideKc2bVPezgcQZXRdq8RUZHPUQCLcB/s1600/hashmap-infinite-loop-thread-2-steps-summary.png)

### Thread 1 again got a chance for execution.

Now Thread 1 will resume transfer() process, but it will end up Nodes in Infinte loop.  
This happen becaused Thread 2 has actually reverse the Node links. 

[](https://3.bp.blogspot.com/-HqB6Z00Cqz0/WTQD7Pkmj7I/AAAAAAAACA4/W2Roiby_XXY0U3Uyx9omDQer1T5-DA0WgCLcB/s1600/hashmap-infinite-loop-thread-1-steps-summary-after.png)

**Any further request for get/put will end up in Infinite loop.**

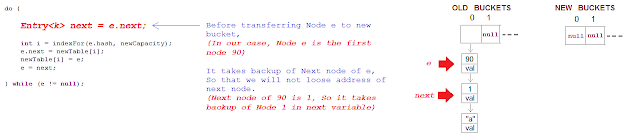
### Still not clear how it ends up in Infinite loop?

### Lets understand each step of both Threads in detailed going through Algorithm step by step.

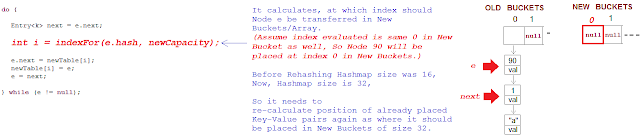
### 

### Before going into details Lets understand what transfer() method does:

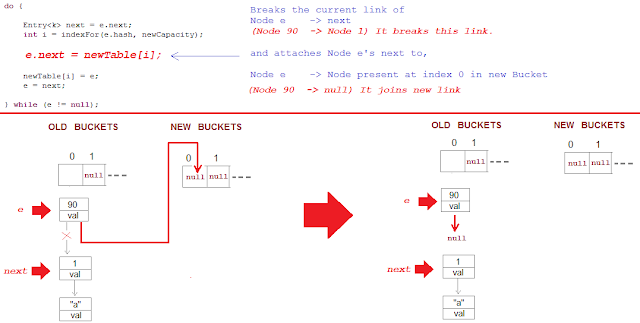
**STEP 1:  Entry<k> next = e.next;**

[](https://2.bp.blogspot.com/-TrQ6o7Z4hxo/WS-o4jTn5CI/AAAAAAAAB5g/fPGPlSRNlHIrBR5M21gkoY0bGMuKTXkJwCLcB/s1600/infinite-loop-in-hashmap-rehashing-code-explained-step-1.png)

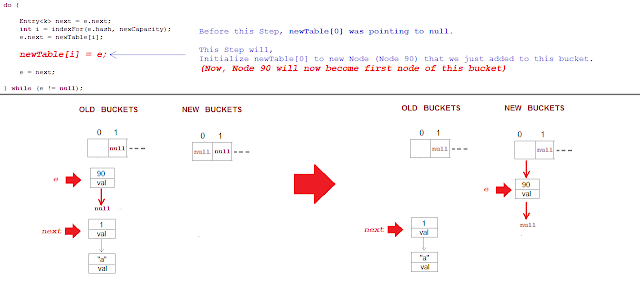
**STEP 2:  int i = indexFor(e.hash, newCapacity);**

[](https://2.bp.blogspot.com/-eXmR5qzDSbQ/WS-pA5eaTsI/AAAAAAAAB5o/uhDnNfnyigsMg4hTnsdnh5DkKN247V1GACLcB/s1600/infinite-loop-in-hashmap-rehashing-code-explained-step-2.png)

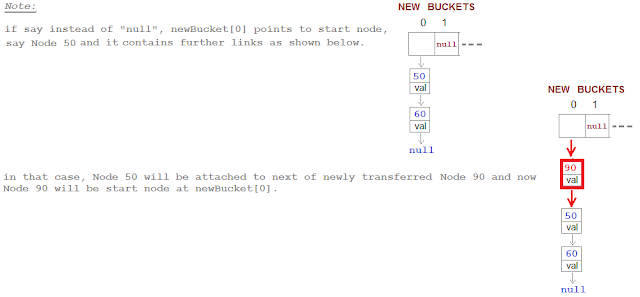
**STEP 3:  e.next = newTable[i];**

[](https://3.bp.blogspot.com/-iwhlwjQEZUM/WS-5-7ZsUuI/AAAAAAAAB6o/93DlM-13fmUM1_lf9Py63do_X9En-9eyQCEw/s1600/infinite-loop-in-hashmap-rehashing-code-explained-step-3.png)

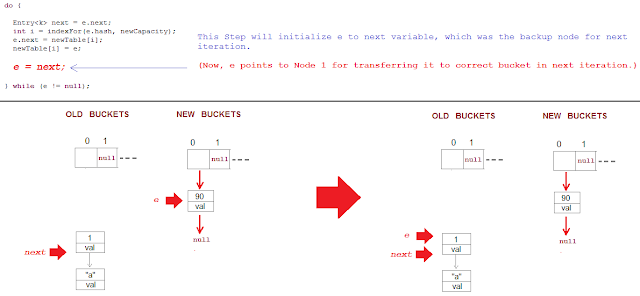
**STEP 4:  newTable[i] = e;**

[](https://4.bp.blogspot.com/-0sMZZN3ryr0/WS-61lfgbnI/AAAAAAAAB6s/v-vIQXkXGSsklxTZp54989sLDXoiAfmNwCLcB/s1600/infinite-loop-in-hashmap-rehashing-code-explained-step-4.png)

**Note:**

[](https://2.bp.blogspot.com/-l8W0g5HUt6A/WS-6-S3NJVI/AAAAAAAAB6w/ldWOimpuRIwOhJTmflHtIR5S47PuZAdHQCLcB/s1600/infinite-loop-in-hashmap-rehashing-code-explained-step-4-part-2.png)

**STEP 5:  e = next;**

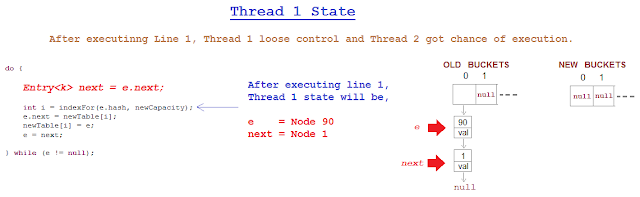
[](https://4.bp.blogspot.com/-PItci23ro5I/WS-9g2O5CaI/AAAAAAAAB7E/qE0V673vOHMRkHJW54OYYZmmhaPn2AYlwCLcB/s1600/infinite-loop-in-hashmap-rehashing-code-explained-step-5.png)

### Too much code.... Now let's start with Race condition....

**Thread 1 Steps.**

**Thread 1 got a chance for execution, and it executed below steps,**

1. Thread 1 try to put 13th key-value pair,
2. Thread 1 founds that Threshold limit is reached and it creates new Buckets of increased capacity. **So map's capacity is increased from 16 to 32.**
3. Thread 1 now starts the transfer process for transferring key-value pairs present at bucket 0 from old array to new array at bucket 0   
   (Assume index evaluated for storing key-value pairs in new array is same that is index 0).  
     
   For transferring, it calls the transfer() method and enters in loop.
4. After executing first line inside loop(in transfer() method), Thread 1 points to first key-value pair and next(second) key-value pair to start transfer process.     
     
   **Before it execute any steps, Thread 1 loose the control and Thread 2 got chance for execution.**
5. So, Current state of Thread 1 is,  **e = Node 90**and **next = Node 1.**

[](https://1.bp.blogspot.com/-EIbGholCJnY/WS_3ecbIwVI/AAAAAAAAB70/VGrRsswStzwP4kUKmIp1KkrIkys0SPSmACLcB/s1600/infinite-loop-thread-1-state.png)

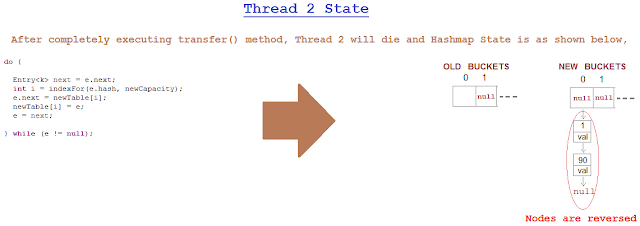
**Thread 1 after pointing to key-value pairs and before starting the transfer process, loose the control and Thread 2 got a chance for execution.**

**Thread 2 Steps.**

**Thread 2 got a chance for execution, and it executed below steps,**

1. Thread 2 try to put 13th key-value pair,
2. Thread 2 founds that Threshold limit is reached and it creates new Buckets of increased capacity. **So map's capacity is increased from 16 to 32.**
3. Thread 2 now starts the transfer process for transferring key-value pairs present at bucket 0 from old array to new array at bucket 0   
   (Assume index evaluated for storing key-value pairs in new array is same that is index 0).  
     
   For transferring, it calls the transfer() method and enters in loop.
4. Thread 2 points to first key-value pair and next(second) key-value pair to start transfer process.
5. Fortunately, Thread 2 executed complete transfer() method without loosing control to other Thread.
6. While transferring key-value pairs from old buckets to new buckets, key-value pairs will be reversed in New buckets because hashmap will add key-value pairs at the start and not at the end. It does this to avoid traversing linked list every time and keep constant performance.
7. **Thread 2 will transfer all key-value pairs from Old buckets to New buckets and Thread 1 will get chance for execution.**

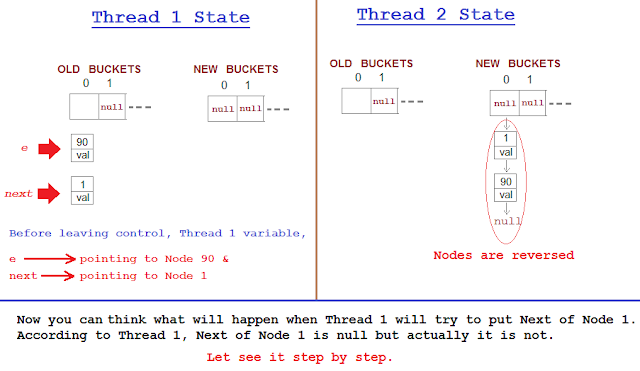
**After Thread 2 execution, Hashmap state will be as shown below,**

[](https://2.bp.blogspot.com/-FaJfXFvHLWE/WTAC5BM08xI/AAAAAAAAB8M/5HOjoZKwIWQEckkwYZNOevBtGeMP8AfmACLcB/s1600/infinite-loop-thread-2-state.png)

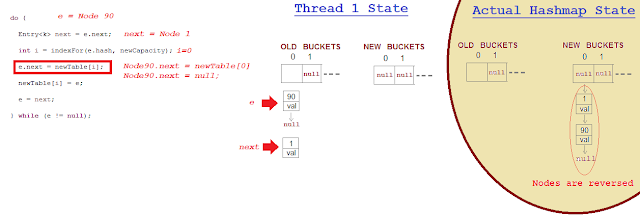
**Thread 1 Steps.**

**Thread 1 now got a chance for execution.**

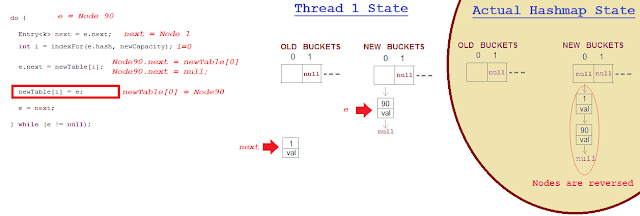
**Let see how Hashmap looks like now from Thread 1 perspective and in actual how it is.**

[](https://4.bp.blogspot.com/-KWr6i_WmsWQ/WTAoZ6McOlI/AAAAAAAAB8k/Oyt8vmznnxcjNZG8H6HZdJyoBSdxW5cLwCLcB/s1600/infinite-loop-thread-1-state-part-1.png)

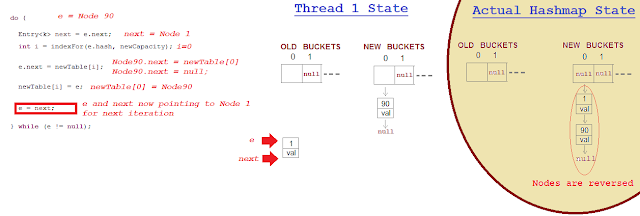
1. **Thread 1 will execute,  int i = indexFor(e.hash, newCapacity);  & e.next = newTable[i];    
   Hashmap will look like below.**

**[](https://1.bp.blogspot.com/-sBPulQx1y3E/WTAt71NrkgI/AAAAAAAAB88/YESwaWOwGwo436PZQ11XfRDyKGaZ0gIwACLcB/s1600/infinite-loop-thread-1-state-part-2.png)**

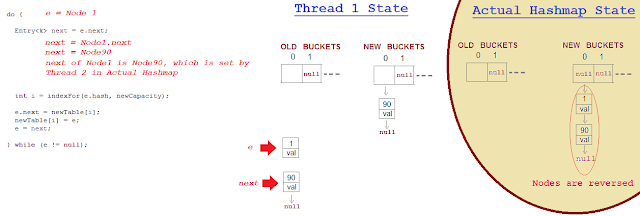
1. **Thread 1 will execute,  newTable[i] = e;  , Hashmap will look like below.**

**[](https://3.bp.blogspot.com/-1GpFdgyD9Is/WTA3h-Z7XfI/AAAAAAAAB9Y/ACTiu08cm8gvUMGeGsKsjLio95XPbDXJACEw/s1600/infinite-loop-thread-1-state-part-3.png)**

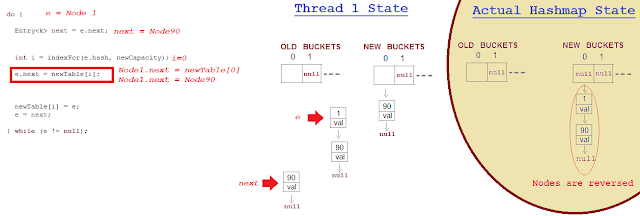
1. **Thread 1 will execute,  e = next; , Hashmap will look like below.**

[](https://3.bp.blogspot.com/-CvN0GOwoeGk/WTA536Zvh5I/AAAAAAAAB9o/PIsG7E4OigYC6UevcSt31AfDiGZ9fEivQCLcB/s1600/infinite-loop-thread-1-state-part-4.png)

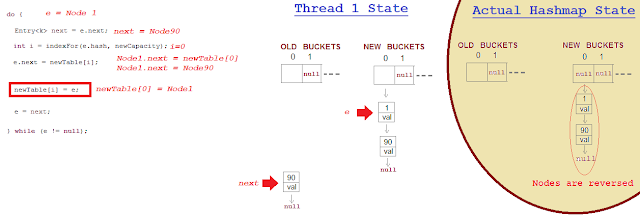
1. **Thread 1 will execute, Entry next = e.next  , Hashmap will look like below.**

**[](https://3.bp.blogspot.com/-mTqlCqf1L8E/WTBA9Bi1UnI/AAAAAAAAB-A/UffRH6aXZZwI-m3-IkyyxvMTMGq5cfi4ACEw/s1600/infinite-loop-thread-1-state-part-5.png)**

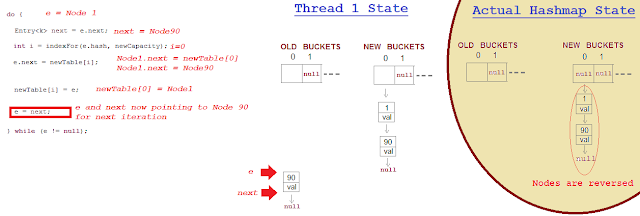
1. **Thread 1 will execute,  int i = indexFor(e.hash, newCapacity);  & e.next = newTable[i]; Hashmap will look like below.**

**[](https://2.bp.blogspot.com/-giBRJ0gn7L4/WTBUFxB-JuI/AAAAAAAAB-Y/3_HhgtV52JQyC0Afyw7005q6EJNCNHuVwCLcB/s1600/infinite-loop-thread-1-state-part-6.png)**

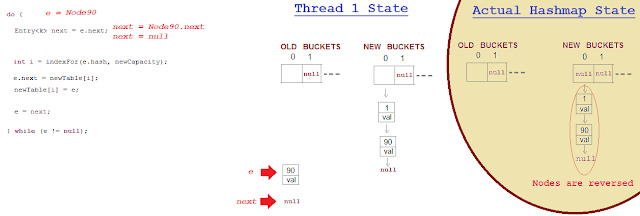
1. **Thread 1 will execute,  newTable[i] = e;  , Hashmap will look like below.**

**[](https://2.bp.blogspot.com/-1_KHg9B70-k/WTB0MYxUYSI/AAAAAAAAB_4/u-5XvMScPSAo3y-NaRGxYzQIvgvG2gMYgCLcB/s1600/infinite-loop-thread-1-state-part-7.png)**

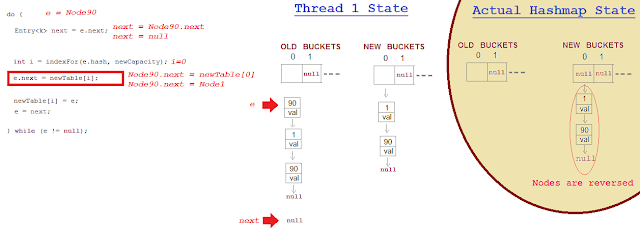
1. **Thread 1 will execute,  e = next;  , Hashmap will look like below.**

[](https://1.bp.blogspot.com/-8D5E9rvzu5c/WTBWSdPIDeI/AAAAAAAAB-k/ulIiNfPiaqcsakEIxwzUWeN2tyUde14JQCLcB/s1600/infinite-loop-thread-1-state-part-8.png)

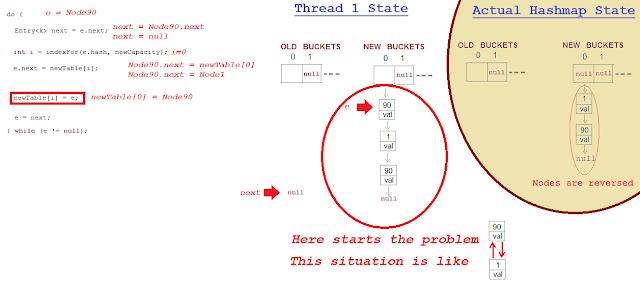
1. **Thread 1 will execute, Entry next = e.next , Hashmap will look like below.**

[](https://3.bp.blogspot.com/-4GGrFKk3N8g/WTBeAnlreeI/AAAAAAAAB_M/RhLy6ZPSQqkulKnVahbYNZdL_Kv742oZgCLcB/s1600/infinite-loop-thread-1-state-part-9.png)

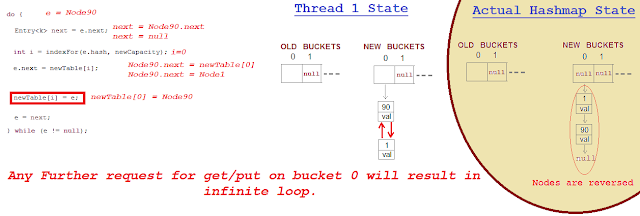
1. **Thread 1 will execute,  int i = indexFor(e.hash, newCapacity);  & e.next = newTable[i]; Hashmap will look like below.**

[](https://2.bp.blogspot.com/-mljY6Hf4Eeg/WTBe8NPDI6I/AAAAAAAAB_U/QNaLfRiaiSAiegiQOjHUiLl7pCKRQbVNACLcB/s1600/infinite-loop-thread-1-state-part-10.png)

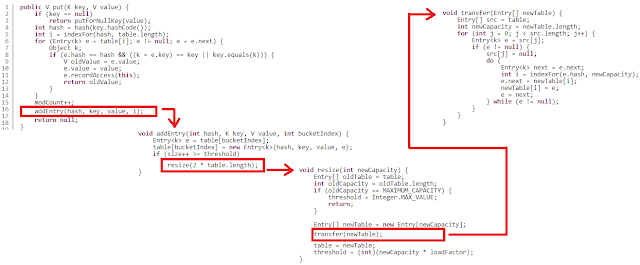
1. **Thread 1 will execute,  e = next; , Hashmap will look like below.**

[](https://2.bp.blogspot.com/-zrl-AafTBmA/WTBgqhGjUsI/AAAAAAAAB_c/sH08T8yNy2ADIqbWXBbcjv4_S0J72XDfgCLcB/s1600/infinite-loop-thread-1-state-part-11.png)

1. **What will happen if any get or put request will come now on index 0; That request will go in infinite loop.**

**[](https://1.bp.blogspot.com/-KBBjUfeepQU/WTBiGMpUr-I/AAAAAAAAB_g/pJSdA1cvcAMwA-fBm5T65mcrAE2RUYWggCEw/s1600/infinite-loop-thread-1-state-part-12.png)**

### HashMap Infinite loop Method call Stack trace

[](https://1.bp.blogspot.com/-acrUyL8GFJM/WS_Cl0IeU2I/AAAAAAAAB7c/t6pFELExz00bgb04q9fiPgJwPgMvYXd4QCLcB/s1600/hashmap-infinite-loop-method-call-stack-trace.png)