How HashMap Works – 2022

Case-1. **Simple object with faulty equals() and hashCode() overridden, two instances of same object.**

@Data  
@AllArgsConstructor  
**public class** Emp {  
 **private** String **name**;  
  
 *// Faulty Implementation of Equals and Hashcode* @Override  
 **public boolean** equals(Object obj) {  
 **return true**;  
 }  
  
 @Override  
 **public int** hashCode() {  
 **return** 11;  
 }  
}

**public static void** main(String[] args) {  
 Emp e1 = **new** Emp(**"John"**);  
 Emp e2 = **new** Emp(**"Vidya"**);  
  
 HashMap<Emp, String> map = **new** HashMap<>();  
 map.put(e1, **"A"**);  
 map.put(e2, **"B"**);  
 System.***out***.println(map);  
}

**OUTPUT**

{Emp(name=John)=B}

It means that when you first entered the key, the key will be stored which will not be replaced. When you entered the second key, Map internally checks whether it is the same key or not as per the rules, if it is the same, then it replaces the value but not the key. That is why you got the weird result. The above output gives surprising result. If you observe the above Emp class, you can observe that we have overridden the hashcode() method and while overidding the equals() method we are returning true value. What will happen if we return the false value for the equals() method ? Let us see below.

@Override  
**public boolean** equals(Object obj) {  
 **return false**;  
}

**OUTPUT**

{Emp(name=John)=A, Emp(name=Vidya)=B}

Here you can see the output is correct. It means that even if hashCode is same for all the objects, the equals method returns false, it means objects are apprently different from each other. So the baseline is that if the hashcodes for the objects are same , still objects are different by the equality. While retrieving, java compares the key's equals method into consideration, that is why we override equals() and hashCode() method.

**How HashMAp works in Java**

HashMap works on principle of hashing, we have put () and get () method for storing and retrieving object form hashMap.When we pass an both key and value to put() method to store on HashMap, it uses key object hashcode() method to calculate hashcode and they 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.

**How does java Hashmap work internally**

**What is Hashing?**  
Hashing in its simplest form, is a way to assigning a unique code for any variable/object after applying any formula/algorithm on its properties. A true Hashing function must follow this rule:  
  
Hash function should return the same hash code each and every time, when function is applied on same or equal objects. In other words, two equal objects must produce same hash code consistently.  
  
Note: All objects in java inherit a default implementation of hashCode() function defined in Object class. This function produce hash code by typically converting the internal address of the object into an integer, thus producing different hash codes for all different objects.

HashMap is an array of Entry objects  
  
Have a look what this Object is:  
static class Entry<K,V> implements Map.Entry<K,V> {

final K key;

V value;

Entry<K,V> next;

final int hash;

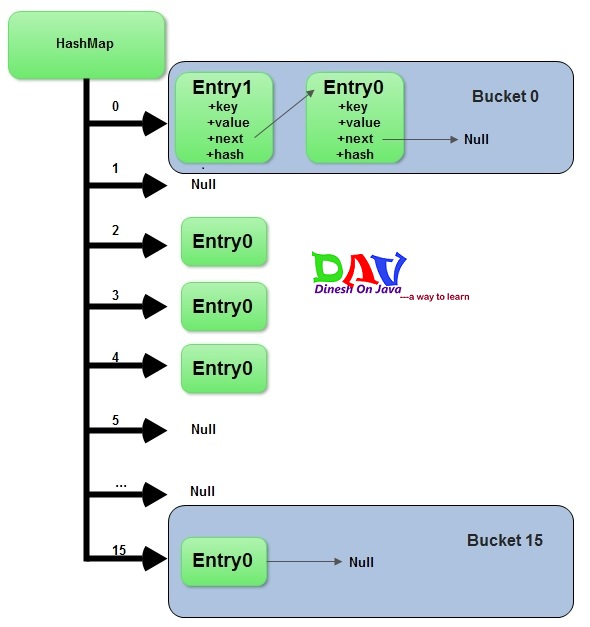
...

}

Each Entry object represents key-value pair. Field next refers to other Entry object if a bucket has more than 1 Entry. Sometimes it might happen that hashCodes for 2 different objects are the same. In this case 2 objects will be saved in one bucket and will be presented as LinkedList. The entry point is more recently added object. This object refers to other object with next field and so one. Last entry refers to null.  
When you create HashMap with default constructor

**HashMap hashMap = new HashMap();**

**Array is gets created with size 16 and default 0.75 load balance.**



**Adding a new key-value pair**

1. Calculate ***hashcode***for the key
2. Calculate position hash % (arrayLength-1)) where element should be placed(bucket number)
3. If you try to add a value with a key which has already been saved in ***HashMap***, then value gets overwritten.
4. Otherwise element is added to the bucket. If bucket has already at least one element - a new one is gets added and placed in the first position in the bucket. Its next field refers to the old element.

**Deletion:**

1. Calculate ***hashcode***for the given key
2. Calculate bucket number (hash % (arrayLength-1))
3. Get a reference to the first Entry object in the bucket and by means of equals method iterate over all entries in the given bucket. Eventually we will find correct Entry. If desired element is not found - return null

**What put() method actually does**  
Before going into put() method’s implementation, it is very important to learn that instances of Entry class are stored in an array.HashMap class defines this variable as:

**transient** Entry[] table;

public V put(K key, V value) {

if (key == null)

return putForNullKey(value);

int hash = hash(key.hashCode());

int i = indexFor(hash, table.length);

for (Entry<k , V> e = table[i]; e != null; e = e.next) {

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

V oldValue = e.value;

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

modCount++;

addEntry(hash, key, value, i);

return null;

}

**Lets note down the steps one by one:**  
  
**Step1-** First of all, key object is checked for null. If key is null, value is stored in table[0] position. Because hash code for null is always 0.  
  
**Step2**- Then on next step, a hash value is calculated using key’s hash code by calling its **hashCode()** method. This hash value is used to calculate index in array for storing Entry object. JDK designers well assumed that there might be some poorly written***hashCode()*** functions that can return very high or low hash code value. To solve this issue, they introduced another hash() function, and passed the object’s hash code to this hash() function to bring hash value in range of array index size.  
  
**Step3-**Now***indexFor(hash, table.length)*** function is called to calculate exact index position for storing the Entry object.  
  
**Step4**- Here comes the main part. Now, as we know that two unequal objects can have same hash code value, how two different objects will be stored in same array location [called bucket].  
  
Answer is **LinkedList**. If you remember, Entry class had an attribute “next”. This attribute always points to next object in chain. This is exactly the behavior of LinkedList.  
  
So, in case of collision, Entry objects are stored in LinkedList form. When an Entry object needs to be stored in particular index, HashMap checks whether there is already an entry?? If there is no entry already present, Entry object is stored in this location.  
  
If there is already an object sitting on calculated index, its next attribute is checked. If it is null, and current Entry object becomes next node in LinkedList. If next variable is not null, procedure is followed until next is evaluated as null.  
  
What if we add the another value object with same key as entered before. Logically, it should replace the old value. How it is done? Well, after determining the index position of Entry object, while iterating over LinkedList on calculated index, HashMap calls equals method on key object for each Entry object. All these Entry objects in LinkedList will have similar hash code but equals() method will test for true equality. If key.equals(k) will be true then both keys are treated as same key object. This will cause the replacing of value object inside Entry object only.  
  
In this way, HashMap ensure the uniqueness of keys.  
  
**How *get()* methods works internally**  
Now we have got the idea, how key-value pairs are stored in HashMap. Next big question is : what happens when an object is passed in get method of HashMap? How the value object is determined?  
  
Answer we already should know that the way key uniqueness is determined in put() method , same logic is applied in get() method also. The moment HashMap identify exact match for the key object passed as argument, it simply returns the value object stored in current Entry object.  
  
If no match is found, get() method returns null.  
  
Let have a look at code:

public V get(Object key) {

if (key == null)

return getForNullKey();

int hash = hash(key.hashCode());

for (Entry<k , V> e = table[indexFor(hash, table.length)]; e != null; e = e.next) {

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

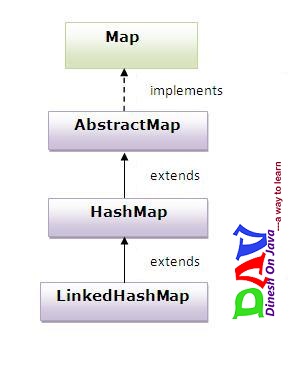
return e.value;

}

return null;

}

**Hierarchy of *LinkedHashMap*class:**



**Methods of HashMap**

**public** V put(K key, V value) {

**if** (table == *EMPTY\_TABLE*) {

inflateTable(threshold);

}

**if** (key == **null**)

**return** putForNullKey(value);

**int** hash = hash(key);

**int** i = *indexFor*(hash, table.length);

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

Object k;

**if** (e.hash == hash && ((k = e.key) == key || **key.equals(k)))** {

V oldValue = e.value;

e.value = value;

e.recordAccess(**this**);

**return** oldValue;

}

}

modCount++;

addEntry(hash, key, value, i);

**return** **null**;

}

**private** V putForNullKey(V value) {

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

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

V oldValue = e.value;

e.value = value;

e.recordAccess(**this**);

**return** oldValue;

}

}

modCount++;

addEntry(0, **null**, value, 0);

**return** **null**;

}

**void** resize(**int** newCapacity) {

Entry[] oldTable = table;

**int** oldCapacity = oldTable.length;

**if** (oldCapacity == *MAXIMUM\_CAPACITY*) {

threshold = Integer.*MAX\_VALUE*;

**return**;

}

Entry[] newTable = **new** Entry[newCapacity];

transfer(newTable, initHashSeedAsNeeded(newCapacity));

table = newTable;

threshold = (**int**)Math.*min*(newCapacity \* loadFactor, *MAXIMUM\_CAPACITY* + 1);

}

**public** V get(Object key) {  
 Node<K,V> e;  
 **return** (e = getNode(*hash*(key), key)) == **null** ? **null** : e.**value**;  
}

**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**;

}

**final** Entry<K,V> getEntry(Object key) {

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

**return** **null**;

}

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

**for** (Entry<K,V> e = table[*indexFor*(hash, table.length)];

e != **null**;

e = e.next) {

Object k;

**if** (e.hash == hash &&

((k = e.key) == key || (key != **null** && key.equals(k))))

**return** e;

}

**return** **null**;

}

/\*\*

\* The default initial capacity - MUST be a power of two.

\*/

**static** **final** **int** *DEFAULT\_INITIAL\_CAPACITY* = 16;

/\*\*

\* The maximum capacity, used if a higher value is implicitly specified

\* by either of the constructors with arguments.

\* MUST be a power of two <= 1<<30.

\*/

**static** **final** **int** *MAXIMUM\_CAPACITY* = 1 << 30;

**HashMap improvements in Java 8**

As part of the work for [JEP 180](https://openjdk.java.net/jeps/180), there is a performance improvement for HashMap objects where there are lots of collisions in the keys by using balanced trees rather than linked lists to store map entries. The principal idea is that once the number of items in a hash bucket grows beyond a certain threshold, that bucket will switch from using a linked list of entries to a balanced tree. In the case of high hash collisions, this will improve worst-case performance from O(n) to O(log n).

Basically when a bucket becomes too big (**currently: TREEIFY\_THRESHOLD = 8**), HashMap dynamically replaces it with an ad-hoc implementation of the treemap. This way rather than having pessimistic O(n) we get much better O(log n).

**How TreeMap Works**

Treemap main advantage is that it allows to store the key-value mappings in a sorted order. Treemap internally uses red black tree.

From the javadocs:

A Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's Introduction to Algorithms.

**Red-black** tree from Wiki:

A red–black tree is a type of self-balancing binary search tree, a data structure used in computer science. The self-balancing is provided by painting each node with one of two colors (these are typically called 'red' and 'black', hence the name of the trees) in such a way that the resulting painted tree satisfies certain properties that don't allow it to become significantly unbalanced. When the tree is modified, the new tree is subsequently rearranged and repainted to restore the coloring properties. The properties are designed in such a way that this rearranging and recoloring can be performed efficiently. The balancing of the tree is not perfect but it is good enough to allow it to guarantee searching in O(log n) time, where n is the total number of elements in the tree. The insertion, and deletion operations, along with the tree rearrangement and recoloring are also performed in O(log n)

Basic source code for treemap is given below.

final int compare(Object k1, Object k2) {

return comparator==null ? ((Comparable<? super K>)k1).compareTo((K)k2)

: comparator.compare((K)k1, (K)k2);

}

public V put(K key, V value) {

Entry<K,V> t = root;

if (t == null) {

compare(key, key); // type (and possibly null) check

root = new Entry<>(key, value, null);

size = 1;

modCount++;

return null;

}

int cmp;

Entry<K,V> parent;

// split comparator and comparable paths

Comparator<? super K> cpr = comparator;

if (cpr != null) {

do {

parent = t;

cmp = cpr.compare(key, t.key);

if (cmp < 0)

t = t.left;

else if (cmp > 0)

t = t.right;

else

return t.setValue(value);

} while (t != null);

}

else {

if (key == null)

throw new NullPointerException();

Comparable<? super K> k = (Comparable<? super K>) key;

do {

parent = t;

cmp = k.compareTo(t.key);

if (cmp < 0)

t = t.left;

else if (cmp > 0)

t = t.right;

else

return t.setValue(value);

} while (t != null);

}

Entry<K,V> e = new Entry<>(key, value, parent);

if (cmp < 0)

parent.left = e;

else

parent.right = e;

fixAfterInsertion(e);

size++;

modCount++;

return null;

}

public V get(Object key) {

Entry<K,V> p = getEntry(key);

return (p==null ? null : p.value);

}

final Entry<K,V> getEntry(Object key) {

// Offload comparator-based version for sake of performance

if (comparator != null)

return getEntryUsingComparator(key);

if (key == null)

throw new NullPointerException();

Comparable<? super K> k = (Comparable<? super K>) key;

Entry<K,V> p = root;

while (p != null) {

int cmp = k.compareTo(p.key);

if (cmp < 0)

p = p.left;

else if (cmp > 0)

p = p.right;

else

return p;

}

return null;

}

final Entry<K,V> getEntryUsingComparator(Object key) {

K k = (K) key;

Comparator<? super K> cpr = comparator;

if (cpr != null) {

Entry<K,V> p = root;

while (p != null) {

int cmp = cpr.compare(k, p.key);

if (cmp < 0)

p = p.left;

else if (cmp > 0)

p = p.right;

else

return p;

}

}

return null;

}

**HashSet**

public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, java.io.Serializable

{

**private transient HashMap<E,Object> map;**

private static final Object PRESENT = new Object();// Dummy value to associate with an Object in the backing Map

public HashSet() {

map = new HashMap<E,Object>();

}

public Iterator<E> iterator() {

return map.keySet().iterator();

}

public boolean add(E e) {

return map.put(e, PRESENT)==null;

}

public boolean remove(Object o) {

return map.remove(o)==PRESENT;

}

private void writeObject(java.io.ObjectOutputStream s)

throws java.io.IOException {

// Write out any hidden serialization magic

s.defaultWriteObject();

s.writeInt(map.capacity());// Write out HashMap capacity and load factor

s.writeFloat(map.loadFactor());

s.writeInt(map.size());// Write out size

for (Iterator i=map.keySet().iterator(); i.hasNext(); ) // Write out all elements in the proper order.

s.writeObject(i.next());

}

private void readObject(java.io.ObjectInputStream s)

throws java.io.IOException, ClassNotFoundException {

// Read in any hidden serialization magic

s.defaultReadObject();

int capacity = s.readInt();// Read in HashMap capacity and load factor and create backing HashMap

float loadFactor = s.readFloat();

map = (((HashSet)this) instanceof LinkedHashSet ?

new LinkedHashMap<E,Object>(capacity, loadFactor) :

new HashMap<E,Object>(capacity, loadFactor));

// Read in size

int size = s.readInt();

for (int i=0; i<size; i++) { // Read in all elements in the proper order.

E e = (E) s.readObject();

map.put(e, PRESENT);

}

}

}

**TreeSet**

public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>, Cloneable, java.io.Serializable

{

**private transient NavigableMap<E,Object> m;**

// Dummy value to associate with an Object in the backing Map

private static final Object PRESENT = new Object();

public TreeSet(Comparator<? super E> comparator) {

this(new TreeMap<E,Object>(comparator));

}

public Iterator<E> iterator() {

return m.navigableKeySet().iterator();

}

public Iterator<E> descendingIterator() {

return m.descendingKeySet().iterator();

}

public NavigableSet<E> descendingSet() {

return new TreeSet(m.descendingMap());

}

public boolean add(E e) {

return m.put(e, PRESENT)==null;

}

public boolean remove(Object o) {

return m.remove(o)==PRESENT;

}

public NavigableSet<E> headSet(E toElement, boolean inclusive) {

return new TreeSet<E>(m.headMap(toElement, inclusive));

}

public NavigableSet<E> tailSet(E fromElement, boolean inclusive) {

return new TreeSet<E>(m.tailMap(fromElement, inclusive));

}

public SortedSet<E> subSet(E fromElement, E toElement) {

return subSet(fromElement, true, toElement, false);

}

public E first() {

return m.firstKey();

}

public E last() {

return m.lastKey();

}

public E lower(E e) {

return m.lowerKey(e);

}

public E floor(E e) {

return m.floorKey(e);

}

public E ceiling(E e) {

return m.ceilingKey(e);

}

public E higher(E e) {

return m.higherKey(e);

}

private void writeObject(java.io.ObjectOutputStream s)

throws java.io.IOException {

// Write out any hidden stuff

s.defaultWriteObject();

// Write out Comparator

s.writeObject(m.comparator());

// Write out size

s.writeInt(m.size());

// Write out all elements in the proper order.

for (Iterator i=m.keySet().iterator(); i.hasNext(); )

s.writeObject(i.next());

}

private void readObject(java.io.ObjectInputStream s)

throws java.io.IOException, ClassNotFoundException {

// Read in any hidden stuff

s.defaultReadObject();

// Read in Comparator

Comparator<? super E> c = (Comparator<? super E>) s.readObject();

// Create backing TreeMap

TreeMap<E,Object> tm;

if (c==null)

tm = new TreeMap<E,Object>();

else

tm = new TreeMap<E,Object>(c);

m = tm;

// Read in size

int size = s.readInt();

tm.readTreeSet(size, s, PRESENT);

}

}

**LinkedHashSet**

public class LinkedHashSet<E> extends HashSet<E> implements Set<E>, Cloneable, java.io.Serializable

{

public LinkedHashSet(int initialCapacity, float loadFactor) {

super(initialCapacity, loadFactor, true);

}

public LinkedHashSet(int initialCapacity) {

super(initialCapacity, .75f, true);

}

public LinkedHashSet() {

super(16, .75f, true); // This internally calls the following of the HashSet class

//HashSet(int initialCapacity, float loadFactor, boolean dummy) {

//map = new LinkedHashMap<E,Object>(initialCapacity, loadFactor);

//}

}

}

**LinkedHashMap Implementation in Java**

*LinkedHashMap*is one of the most commonly used *Map*implementation in Java. *LinkedHashMap* inherits from*HashMap*and implements Map interface. So basic building block is same as that of HashMap. It will be appropriate to over *HashMap* internals before delving into details of *LinkedHashMap*. I have already discussed HashMap internals[here](http://geekrai.blogspot.in/2013/06/hashmap-implementation-in-java.html).  This post talks about the delta which *LinkedHashMap*adds on top of *HashMap*.

**Building Blocks**

*LinkedHashMap*uses same array structure as used by *HashMap*. This means that it stores data in the array of Entry objects. Let's cover major aspects of *LinkedHashMap*.

***Entry* :**HashMap stores objects in random order. Even in LinkedHashMap objects are put in the same way; but with some extra overheads all these objects are interconnected to maintain a particular order. Let's see how Entry object differs here with respect to HashMap Entry.

**static class Entry<K,V> extends HashMap.Entry<K,V>{**

**Entry<K,V> before, after;**

**Entry(int hash, K key, V value, HashMap.Entry<K,V) next){**

**super(hash,key,value,next);**

**}**

**}**

So LinkedHashMap's Entry class adds two extra pointers *before*and *after*to form a doubly linked list. *HashMap.Entry* is defined in HashMap internals [post](http://geekrai.blogspot.in/2013/06/hashmap-implementation-in-java.html).  
  
***Header:***Stores the head of the doubly linked list. Notice the way it is getting initialized/constructed at below snippet. *This means that header is both start as well as end of the doubly linked list.*  
  
       **Entry<K,V> header;**  
**//construction or initialization**  
**header = new Entry<>(-1, null, null, null);**  
**header.before = header.after = header;**  
 ***AccessOrder :***This defines the order in which you can iterate a *LinkedHashMap*. It supports access-order as well as insertion-order. Default constructor supports insertion order (as shown below).  
            
        **boolean accessOrder;**  
 **public LinkedHashMap(){**  
**super();**  
**accessOrder = false;**  
**}**  
  
Other aspects like size, load factor, capacity, threshold etc remain unchanged.

**Introspection**

Let's do similar exercise as done in HashMap post, to store fruits inside LinkedHashMap and introspect internal details.

**public class LinkedHashMapSimul {**

**static Map<String, Integer> map;**

**// Actual Method from HashMap library**

**static int indexFor(int h, int length) {**

**return h & (length - 1);**

**}**

**// Actual Method from HashMap library**

**static int hash(int h) {**

**h ^= (h >>> 20) ^ (h >>> 12);**

**return h ^ (h >>> 7) ^ (h >>> 4);**

**}**

**public static void interrospect() {**

**try {**

**Class<?> c = map.getClass();**

**c = c.getSuperclass();**

**Field f = c.getDeclaredField("size");**

**f.setAccessible(true);**

**int size = f.getInt(map);**

**f = c.getDeclaredField("loadFactor");**

**f.setAccessible(true);**

**float loadFactor = f.getFloat(map);**

**f = c.getDeclaredField("threshold");**

**f.setAccessible(true);**

**int threshold = f.getInt(map);**

**f = c.getDeclaredField("modCount");**

**f.setAccessible(true);**

**int modCount = f.getInt(map);**

**Method m = c.getDeclaredMethod("capacity", null);**

**m.setAccessible(true);**

**int capacity = (Integer) m.invoke(map, null);**

**System.out.println("Size: " + size + "; LoadFactor: " + loadFactor**

**+ "; Threashold: " + threshold + "; ModCount: " + modCount**

**+ " ; capacity :" + capacity);**

**} catch (SecurityException e) {**

**e.printStackTrace();**

**} catch (NoSuchFieldException e) {**

**e.printStackTrace();**

**} catch (IllegalArgumentException e) {**

**e.printStackTrace();**

**} catch (IllegalAccessException e) {**

**e.printStackTrace();**

**} catch (NoSuchMethodException e) {**

**e.printStackTrace();**

**} catch (InvocationTargetException e) {**

**e.printStackTrace();**

**}**

**}**

**public static void main(String[] args) {**

**map = new LinkedHashMap<String, Integer>();**

**map.put("guava", indexFor(hash("guava".hashCode()), 16));**

**map.put("mango", indexFor(hash("mango".hashCode()), 16));**

**map.put("pear", indexFor(hash("pear".hashCode()), 16));**

**map.put("banana", indexFor(hash("banana".hashCode()), 16));**

**System.out.println(" Map :" + map);**

**interrospect();**

**}**

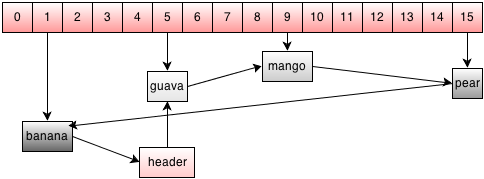
**}**

**Output**

 Map :{guava=5, mango=9, pear=15, banana=1}

Size: 4; LoadFactor: 0.75; Threashold: 12; ModCount: 4 ; capacity :16

So the output confirms that the iteration order is same as insertion. Above class creates default *LinkedHashMap*; which means initial capacity is 16, load factor is 0.75 and accessOrder is false. Below diagram shows the mapping of above items in the Entry array. Note that linked list begins from header and ends at header.

[](http://2.bp.blogspot.com/-Y9Qm6DdfgeI/Ubyo7c2YO2I/AAAAAAAAHAQ/vtczc00RTX0/s1600/LinkedHashMap.png)