**ConcurrentHashMap**

## Data Structure

ConcurrentHashMap main motive is to allow concurrent access to the map. [HashTable](http://javarticles.com/2012/06/hashmap.html" \t "_blank) offers synchronized access to the map but the entire map is locked to perform any sort of operation.

In ConcurrentHashMap, the basic strategy is to subdivide the table among segments so that the lock is applied only on a segment rather than the entire table. Each segment manages its own internal hash table in size 2x >=(capacity/no. of segments).

Locking is applied only for updates. In case of of retrievals, it allows full concurrency, retrievals reflect the results of the most recently completed update operations.

If we assume four threads are going to concurrently work on a map of initial capacity 32, we would want the table to be partitioned into four segments, each managing a hash table of capacity 8.

ConcurrentHashMap Data Structure would be:

|  |
| --- |
| [ConcurrentHashMap](http://i1.wp.com/4.bp.blogspot.com/-bC-jGRWvehA/T9sUqE1-hZI/AAAAAAAAA9A/Oypuh4H_Apc/s1600/ConcurrentHashMapDataSructure.png) |
| ConcurrentHashMap |

For comparison sake, below figure shows the data structure of a HashMap.

|  |
| --- |
| [ConcurrentHashMap](http://i1.wp.com/3.bp.blogspot.com/-BVkWRzmS3Z0/T9sUgPup0mI/AAAAAAAAA80/MWa2MjWbHdk/s1600/HashMapDataStructure.png) |
| ConcurrentHashMap |

## Segment

The collection maintains a list of 16 segments by default, each of which is used to guard (or lock on) a single bucket of the map. This effectively means that 16 threads can modify the collection at a single time (as long as they’re all working on different buckets). This level of concurrency can be increased using the optional concurrencyLevelconstructor argument.

[?](http://javarticles.com/2012/06/concurrenthashmap.html)

|  |  |
| --- | --- |
| 1 | public ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel) |

The maximum size it can go up to is 216. Greater the concurrency level, greater would be the no. of segments and lesser the size of hash table that the segment manages. Using a significantly higher value than you need can waste space and time, and a significantly lower value can lead to thread contention.

## Put Entry

### New Entry

ConcurrentHashMap doesn’t allow NULL values. The key cannot be NULL, it is used to locate the segment index and then the actual bucket within the segment. The key’s hash is re-hashed to strengthen the hash value to defend against poor quality hash functions. This is important as the hash will be used to locate both segment and hash table index. The upper bits will be used to locate the segment index and the lower bits to locate the table index within the segment. This is why re-hashing function is different from that of HashMap as the hash bits need to be spread better to regularize both segment and index locations.

[?](http://javarticles.com/2012/06/concurrenthashmap.html)

|  |  |
| --- | --- |
| 1  2  3  4  5 | private static int hash(int h) {      h += (h <>> 10);      h += (h <>>  6);      h += (h <<   2) + (h <>> 16);  } |
| [ConcurrentHashMap](http://i2.wp.com/1.bp.blogspot.com/-pUarxXbgKH4/T9sU5s1hQdI/AAAAAAAAA9M/BRjSVRvPHyE/s1600/ConcurrentHashMapNewEntry.png) |
| ConcurrentHashMap |

### Put If Absent

ConcurrentMap offers new method putIfAbsent() which is similar to put except that the value will not be overridden if key already exists. This is equivalent to

[?](http://javarticles.com/2012/06/concurrenthashmap.html)

|  |  |
| --- | --- |
| 1  2  3  4 | if (!map.containsKey(key))      return map.put(key, value);  else     return map.get(key); |

except that the action is performed atomically.

containsKey is not synchronized so the above code may cause unexpected behavior. Lets look into the below scenario:

1. Thread A is trying to remove a key, calls remove(key), acquires lock.
2. Thread B tries to execute above code, calls containsKey(key). The key exists as Thread A has not yet released the lock so the old value is returned.
3. Thread A removes the key and releases the lock.

The above scenario proves that the code is not atomic as it returned a key which it thinks exists but actually doesn’t.  
Also, performance wise it is slow, containsKey has to locate the segment and traverse through the table to find the key. Method put needs to do the same traversing to locate the bucket and put the value. The new methodputIfAbsent is a bit faster as it avoids double traversing. It goes through the same internal flow as put, avoids overriding the old value if key already exists and simply returns the old value.

## Class Diagram

Below class diagram shows that ConurrentHashMap is made up of Segments which in turn is made up ofHashEntries.

|  |
| --- |
| [ConcurrentHashMap](http://i2.wp.com/4.bp.blogspot.com/-VRadR-OxLKk/T9sVWiXMxOI/AAAAAAAAA9Y/-dho6ItqV9I/s1600/ConcurrentHashMapClassDiagram.png) |
| ConcurrentHashMap |

## Remove Entry

HashEntry is an immutable class so the next link can’t be adjusted. All entries following removed node stays in the list, but all preceding ones need to be cloned.

[?](http://javarticles.com/2012/06/concurrenthashmap.html)

|  |  |
| --- | --- |
| 1  2  3  4  5 | HashEntry newFirst = e.next;  for (HashEntry p = first; p != e; p = p.next) {      newFirst = new HashEntry(p.key, p.hash, newFirst, p.value);      tab[index]= newFirst;  } |
| [ConcurrentHashMap](http://i0.wp.com/1.bp.blogspot.com/-Fw4mhDB6amQ/T9sVtWam3GI/AAAAAAAAA9k/45e_tCAbhRU/s1600/ConcurrentHashMapRemoveEntry.png) |
| ConcurrentHashMap |

## Re-sizing

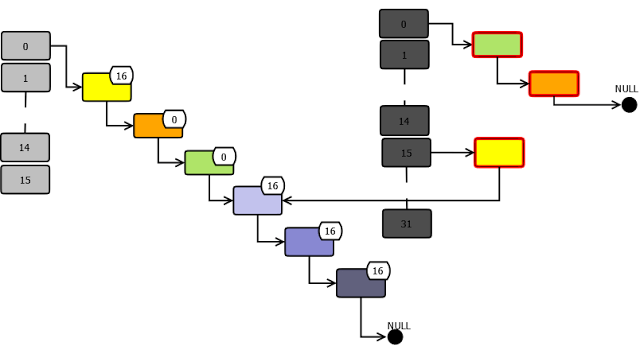
Since power-of-two expansion is used, the elements from each bin must either stay at same index, or move with a power of two offset.

[?](http://javarticles.com/2012/06/concurrenthashmap.html)

|  |  |
| --- | --- |
| 1 | int idx = e.hash & sizeMask; |

For a capacity of 16 elements, sizeMask would be 1111 so the lower 4 bits of hash will form the index.  
When the capacity is doubled, sizeMask would be 11111 and now the lower 5 bits of hash will form the index.

So the index would be same if the 5th is 0 else the index would be incremented by 24.  
The trailing consecutive sequence nodes can be re-used instead of unnecessary node creation if they they point to the same slot after the re-hashing. Here in the below example the trailing nodes in blue hues will all end up in the same index.

[](http://i1.wp.com/2.bp.blogspot.com/-u8JHswAuqzc/T9sV71DN0aI/AAAAAAAAA9w/FYsTc0wZkkI/s1600/ConcurrentHashMapResize.png)

Home

## LinkedHashMap vs. HashMap

LinkedHashMap is a HashMap that also defines the iteration ordering using an additional data structure, a double linked list. By default, the iteration order is same as insertion-order. It can also be the order in which its entries were last accessed so it can be easily extended to build[LRU cache](http://javaopensourcecode.blogspot.in/2012/06/lru-cache.html).

## Data Structure

The data structure of LinkedHashMap extends that of HashMap.

In [HashMap](http://javaopensourcecode.blogspot.in/2012/06/hashmap.html), the data structure is based on array and linked list. An entry finds its location in the array based on its hash value. If an array element is already occupied, the new entry replaces the old entry and the old entry is linked to the new one.

In HashMap, there is no control on the iteration order.

In LinkedHashMap, the iteration order is defined, either by the insertion order or access order.

LinkedHashMap differs from HashMap in that it maintains a doubly-linked list running through all of its entries. The below one is a modified example of the above data structure. It defines the iteration ordering based on the order in which keys were inserted into the map. In order to do so, the entry element is extended to keep track of the after and before element. A zero size LinkedHashMap contains just the Head element with before and after pointing to itself.

|  |
| --- |
| [LinkedHashMap data structure](http://i1.wp.com/1.bp.blogspot.com/-lf_tDJGSjTQ/T9sXJMOrORI/AAAAAAAAA-I/xqkY8G3c1K0/s1600/LinkedHashMapDataStructure.png) |
| LinkedHashMap data strutcture |

Below is the HashMap data structure:

|  |
| --- |
| [HashMap Data Structure](http://i2.wp.com/2.bp.blogspot.com/-5KRTUbSlEy0/T9sWmM9MDsI/AAAAAAAAA98/qBS4y3-a1SU/s1600/HashMapDataStructure.png) |
| HashMap data structure |

## Entry

LinkedHashMap's Entry extends the HashMap's Entry so it also inherits the same properties key, value, hash and the next Entry sharing the index. Other than these, it also has couple of additional properties to maintain the double-linked list, after and before entries.

|  |
| --- |
| [LinkedHashMap Entry class](http://i1.wp.com/3.bp.blogspot.com/-jlesknZ66GE/T9sXjZS4bqI/AAAAAAAAA-g/nnUJN-k3O7E/s1600/LinkedHashMapEntry.png) |
| LinkedHashMap Entry class |

## New Entry

LinkedHashMap inherits HashMap so its internal data structure is same as that of HashMap. Apart from that it also maintains a double-linked list which is circularly linked via the sentinel node called head. Each node contains references to the previous and to the next node . A new node is always added to the end of the list. In order to do so, the last node’s and the header node’s links have to be adjusted.

1. The new node’s next reference will point to the head.
2. The new node’s previous reference will point to the current last node.
3. The current last node’s next reference will point to the new node instead of head.
4. Head’s previous reference will point to the new node.

[?](http://javarticles.com/2012/06/linkedhashmap.html)

|  |  |
| --- | --- |
| 1  2  3  4 | after  = head;  before = head.before;  before.after = this;  after.before = this; |
| [Double linked-list](http://i1.wp.com/2.bp.blogspot.com/-JpROO8qL0E0/T9sXrPfYCYI/AAAAAAAAA-s/P5eL6csaMjM/s1600/LinkedHashMapNewEntry.png) |
| Double linked-list |

Performance is likely to be just slightly below that of HashMap, due to the added expense of maintaining the linked list.

## Access Ordered

A special LinkedHashMap(capacity, loadFactor, accessOrderBoolean) constructor is provided to create a linked hash map whose order of iteration is the order in which its entries were last accessed, from least-recently accessed to most-recently. Invoking the put or get method results in an access to the corresponding entry. If the enclosing Map is access-ordered, it moves the entry to the end of the list; otherwise, it does nothing.

[?](http://javarticles.com/2012/06/linkedhashmap.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | public void testLinkedHashMap() {      LinkedHashMap lru = new LinkedHashMap(16, 0.75f, true);      lru.put("one", null);      lru.put("two", null);      lru.put("three", null);        Iterator itr = lru.keySet().iterator();      while (itr.hasNext()) {          System.out.println(itr.next());      }        System.out.println("\*\* Access one, will move it to end \*\*");      lru.get("one");        itr = lru.keySet().iterator();      while (itr.hasNext()) {          System.out.println(itr.next());      }        System.out.println("\*\* Access two, will move it to end \*\*");      lru.put("two", "two");        itr = lru.keySet().iterator();      while (itr.hasNext()) {          System.out.println(itr.next());      }  } |

[?](http://javarticles.com/2012/06/linkedhashmap.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | Result:  one  two  three  \*\* Access one, will move it to end \*\*  two  three  one  \*\* Access two, will move it to end \*\*  three  one  two |

Thus in access-ordered linked hash maps, merely querying the map with get is a structural modification.

## Iterator

In HashMap, the iterator has to traverse through each table element and the element’s own linked list, requiring time proportional to its *capacity*.  
In LinkedHashMap, it simply has to traverse through its own double-linked list thus requires time proportional to the *size* of the map and not its capacity so HashMap iteration is likely to be more expensive.

## Transfer

Re-sizing is supposed to be faster as it iterates through its double-linked list to transfer the contents into a new table array.

[?](http://javarticles.com/2012/06/linkedhashmap.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | void transfer(HashMap.Entry[] newTable) {      int newCapacity = newTable.length;      for (Entry e = header.after; e != header; e = e.after) {          int index = indexFor(e.hash, newCapacity);          e.next = newTable[index];          newTable[index]= e;      }  } |

## Contains Value

containsValue() is Overridden to take advantage of the faster iterator.

## LRU Cache

If access ordered is true, order of iteration is the order in which its entries were last accessed, from least-recently accessed to most-recently (*access-order*). This kind of map is well-suited to building LRU caches. TheremoveEldestEntry(Entry) method may be overridden to impose a policy for removing stale mappings automatically when new mappings are added to the map. For example,

[?](http://javarticles.com/2012/06/linkedhashmap.html)

|  |  |
| --- | --- |
| 1  2  3 | protected boolean removeEldestEntry(Map.Entry eldest) {      return size() > maxCacheSize;  } |

The eldest entry is returned by header.after. The default implementation of removeEldestEntry() returns false.

## Access Order

The main reason why one prefers LinkedHashMap over HashMap is that it can retain the order in which the elements are accessed. Below is the basic flow that a HashMap goes through to put a new entry. The blue boxes, ‘Add Entry’ and ‘Record Access’ are the ones LinkedHashMap overrides.

|  |
| --- |
| [LinkedHashMap](http://i1.wp.com/3.bp.blogspot.com/-LRZZ17yXKBQ/Uo3hnAan-4I/AAAAAAAADTQ/sJOAuun7Zi0/s1600/hashMap_putEntry.png) |
| LinkedHashMap |

It overrides ‘Record Access’ to record the access order. If the user is interested in the access order, it updates its double linked list.

|  |
| --- |
| [LinkedHashMap](http://i2.wp.com/3.bp.blogspot.com/-XBonkbMFmwU/Uo3ipvtBufI/AAAAAAAADTg/sOrkyTsGotc/s1600/linkedHashMap_RecordAceess.png) |
| LinkedHashMap |

Below entry picture shows how the entry moves up the linked list. Head’s next entry will point to the latest entry accessed.

|  |
| --- |
| [LinkedHashMap](http://i1.wp.com/3.bp.blogspot.com/-UAUOrI0VhSA/Uo3i6YlF31I/AAAAAAAADTo/t7IQ5g7244k/s1600/LinkedHashMap_entry.png) |
| LinkedHashMap |

If E2 is accessed again, HashMap identifies the entry and then calls record access.

The record access is overridden in LinkedHashMap. Below is the class diagram where LinkedHashMap extends HashMap’s entry to override the recordAccess.

|  |
| --- |
| [LinkedHashMap](http://i0.wp.com/2.bp.blogspot.com/-o6jvdvNJ3-A/Uo3kdJwbyiI/AAAAAAAADT8/Je1KsyZPj0Y/s1600/LinkedHashMap_RecordAccessOverride.png) |
| LinkedHashMap |

## Double linked list vs Single Linked List

|  |
| --- |
| [LinkedHashMap](http://i2.wp.com/3.bp.blogspot.com/-psAZ5usCbwE/Uo3lUQiqYfI/AAAAAAAADUE/jbWL-Vbcyzs/s1600/LinkedHashMap_removeEntry.png) |
| LinkedHashMap |

Suppose we want to remove an entry E2.

If we have a single linked list, E3’s next should point to E1 which means if want to eliminate E2, we need to know its previous entry. If it is a single linked list, we will end up traversing the entire list to search the entry previous to E2, thus the performance would in the O(n)

In case of double linked list, the previous pointer in E2 will take us to E3 in O(1).

# HashMap Interview Questions

In this article, we will look into some of the most important HashMap interview questions from the perspective of data structure. Each question takes us to the next one and we will follow it to get to the bottom of the data structure.

1. **Where is the key, value pair stored in HashMap?**

The key and value pair is stored in an array.

1. **How is the index identified?**

The key is used to identify the table index. If we divide the key’s hash code number by the table’s capacity, the remainder will be the table index which is why it is important to override hashCode().

1. **What is collision?**

If two different keys result in the same hash code or different hash codes map to the same table index, there will be a collision. Since two keys can’t sit in the same location, each slot of the bucket array is a pointer to a linked list that contains the key-value pairs hashed to the same location.

1. **What is the maximum capacity of a HashMap?**

The capacity is always in power of 2 irrespective of the size we pass to the constructor. The maximum capacity would be the maximum power of 2 which would be 0100 0000 0000 0000 0000 0000 0000 0000 = 1 << 30

1. **Why is the capacity in power of 2?**

In general, the number of buckets should be prime is so that the hash values are well distributed and will have less collisions. In case of HashMap, the capacity is always a power-of-two. In contrast, Hashtable by default allocates a size of 11, a prime number. If the size is specified, it will create an array of the specified size.

Whereas in HashMap, if the size is specified and is not a power-of-2, it will be incremented to power-of-2 to create the internal array.

In order to convert the hash code to index, we divide it by the capacity and the remainder would be the index. There seems to be a [Bug](http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4631373) in versions from JDK1.4 on-wards where integer and modulus operation are much slower than its earlier versions. If the capacity of array is in power-of-two, the hash code can be easily converted to the index based on a simple AND operation and this seems to be more efficient as compared to modulus operation:

index =  hashCode & (array length-1)

This might be the reason why the array capacity always has to be a power-of-two.

1. **How can a simple AND operation translate the hash code into table index?**

Suppose the hash code=311 and length=16, that is, 24, applying the modulus operation we get:

hash code=length\*quotient + index, or 311 = 16 \* 19 + 7, where table index = 7.

Let us convert the hash code to binary.

311 = 1 0011 0111 in binary

= 1 0011 0111

= 1 0011 0000 + 0111

= 24\*1 + 25\*1 + 26\*0 + 27\*0 + 28\*1 + 0111

= 24(20\*1 + 21\*1 + 22\*0 + 23\*0 + 24\*1) + 0111

= 24(10011) + 0111

This is in the same form as the length\*quotient + index, so table index = 0111.

Thus,we can obtain the table index by masking the hash code using & operation.

table Index = hash code & (2length - 1) = 1 0111 0111 & 1111 = 0111

1. **What effect does table capacity have on the procedure of converting hash code into table index?**

If length=2x, table index = hash code & (2x - 1)

So if hash code=311, table index would be, 1 0111 0111 & 1111 = 0111 which is nothing but the lower xbits of hash code.

Thus, if the table capacity is in power-of-two, only the lower bits of hash code determine the table index.

1. **What is re-hashing in context of HashMap?**

Since HashMap has table capacity in power-of-two, only the lower bits influence the table index. This is why a supplemental hash function is applied to the given hashCode before masking off the low order bits. Its job is to distribute the information over all the bits, and in particular, into the low order bits. This is called re-hashing.

1. **How does put() method of HashMap work in java?**

When the put (key, value) method is called to store the key and value pair on HashMap, it first determines the bucket location where the key and value entry should go.

The implementation calls hashcode() method on the key object to map the hash code on to a table index. Since the table length is in power-of-two and only the lower bits determine the table index, it applies a supplementary hash function on the hash code to further strengthen it. Its on this final hash code that the below AND operation is applied.

Table index = supplementaryHashFunction(key's hash code) & (tableLength -1)

There may be an existing entry for the determined bucket location. In the simplest case, where there is no entry, an entry is created and inserted into the identified bucket.

1. **So what’s the basic algorithm of put method?**

From our last answer, we can summarize it as:

* 1. Get key’s hash code
  2. Apply supplementary hash function on it
  3. Determine table index, based on the new hash code
  4. Insert a new entry of key and value into the identified location

There are many subtle variations to the above algorithm, like, what if the key is null or what if an entry already exists in the bucket location? In the questions that follow, we will discuss these variations.

1. **Does HashMap allow NULL keys?**

Yes. Since key is null, it won’t be able to derive the table index, so it stores the entry into index 0.

1. **What will be the hash code for NULL key?**

Zero.

1. **What happens if the key already exists?**

If key already exists, its value will be replaced by the new one and the old value will be returned to the caller.

1. **What if an entry already exists in the bucket location?**

The old entry will replaced by the new entry and the new entry’s **next**will point to the old entry.

1. **How does get(key) method of HashMap work in java?**

First the key is converted to a table index. There may be 0 or more entries in the form of a linked structure in the bucket found. If an entry already exists then we just need to traverse through the linked structure to find the key’s entry. We know we have found the key’s entry if both the below conditions hold true.

* 1. The key’s hash code and entry’s **hash**code match.
  2. The passed in key and the entry’s **key**are equal

If the hash code doesn’t match then we don’t have to check key’s equality.

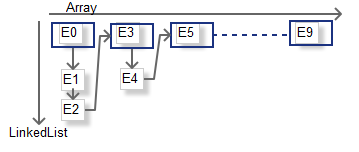
If an entry is found then its **value**will be returned to the caller else a NULL will be returned.

1. **What is the type of the object stored in the array?**

It is of type Entry. Its attributes are key, value, next entry and hash.

1. **Does HashMap retain the order of its elements?**

It doesn’t retain the order. Iterator starts traversing from 0th element and returns the first not null element in the array (E0). Next element would be the one that the array element points to (E1), till it reaches the end of the linked list (E2). Once it reaches the end of linked list, it retrieves the next not null element (E3) from the array and the process goes on it reaches the end of array (E4->E5…E9).

[](http://i2.wp.com/javarticles.com/wp-content/uploads/2012/11/HashMapQADataStructure.png)

*HashMap Data Structure*