HashMap maintains an array of buckets. Each bucket is a linkedlist of key value pairs encapsulated as Entry objects  
   
This array of buckets is called table. Each node of the linked list is an instance of a private class called Entry  
   
transient Entry[] table;  
   
An entry is a private static class inside HashMap which implements Map.Entry  
 

1. **private** **static** **class** Entry<K,V> implements Map.Entry<K,V> {
2. final K key;
3. final **int** hash;
4. V value;
5. Entry<K,V> **next**;
6. }

   
Each entry object exists only for a particular key but values may change (if same key is reinserted later with a different value) - hence key is final while value is not  
   
Each Entry object has a field called next which points to the next Entry thus behaving like a singly linked list.  
   
The hash field stores the hashed value of the key  
  
constructor:  
  
HashMap provides overloaded constructors with parameters for initial capacity and load factor but typically no args constructor is the one most frequently used  
   
default values of these fields are :  
initial capacity : 1 << 4 (ie 16)  
load factor : 0.75  
   
Whenever the element count of the hashmap reaches the load factor fraction of capacity, the map is resized and capacity is doubled  
   
If capacity provided by client is a power of 2, then real capacity will be same as capacity  
else real capacity = nearest power of 2 > provided capacity  
   
maximum capacity is 1<<30 (ie 2 ^30) if capacity provided is greater than that, then real capacity = 2^30  
   
Note that capacity indicates the size of the table array (the array of buckets) and not the number of key-value pairs the HashMap can support  
   
Fail-fast iterator :  
   
HashMap specifications need that iterators should throw ConcurrentMoodificationException if the map contents are changed while a client is iterating over an iterator  
   
This done by keeping track of number of modifications. HashMap has a member int variable named modCount which is incremented everytime the map is altered (any invocation of put(), remove(), putAll() or clear() methods)  
   
A similar field (lets say iteratorModCount) is maintained in the iterator implemetation class. When the iterator is created, the iteratorModCount value is initialized with the same value as HashMap modCount. For every call to any of iterator methods (next(), hasNext() and remove() ) the iteratorModCount is checked against the HashMap modCount. If these two values dont tally, that means HashMap has been modified and ConcurrentModificationException is thrown  
   
When the remove() method of iterator is invoked, after internally calling remove() on the map, the iteratorModCount is reinitialized with the HashMap's modCount  
   
Note: the HashMap's modCount and iterator's modCount fields are neither atomic nor valatile - hence they are vulnerable to interleaving and are not guraranteed to work in a multithreaded context  
  
collections representing keySet and values :  
   
HashMap specifications require that keySet(), values() and entrySet() methods complete with O(1) space complexity - which basically means HashMap cant copy the data into a new collection and return it. Rather these collections have to point to the same location in memory as the actual HashMap contents  
   
This is done by maintaining non-static inner classes called KeySet and EntrySet both of which extend AbstractSet. By virtue of being non static, they implicitly carry a reference to the outer class object (HashMap.this)  
  
[code]    **private** **final** **class** KeySet **extends** AbstractSet<K> {  
        **public** Iterator<K> iterator() {  
            **return** newKeyIterator();  
        }  
        **public** **int** size() {  
            **return** size ;  
        }  
        **public** **boolean** contains(Object o) {  
            **return** containsKey(o);  
        }  
        **public** **boolean** remove(Object o) {  
            **return** HashMap.**this**.removeEntryForKey(o) != **null**;  
        }  
        **public** **void** clear() {  
            HashMap. **this**.clear();  
        }  
    }  
   
    **private** **final** **class** Values **extends** AbstractCollection<V> {  
        **public** Iterator<V> iterator() {  
            **return** newValueIterator();  
        }  
        **public** **int** size() {  
            **return** size ;  
        }  
        **public** **boolean** contains(Object o) {  
            **return** containsValue(o);  
        }  
        **public** **void** clear() {  
            HashMap. **this**.clear();  
        }  
    }[/code]  
  
Now, as we see above, special private classes have been created which internally invoke methods of the outer class.  
   
For the iterators to this classes, special iterator instance is created which iterates over the buckets and entries. This iterator base class HashIterator is again a non static inner class thus having reference to the outer HashMap class through HashMap.this  
  
[code]**private** **abstract** **class** HashIterator <E> **implements** Iterator<E> {  
        Entry<K,V> next;        // next entry to return  
        **int** expectedModCount ;   // For fast-fail  
        **int** index ;              // current slot  
        Entry<K,V> current;     // current entry  
    }  
es the hashed value of the key  
[/code]  
  
Clearly HashIterator needs to keep track of the following:  
1. the current entry  
2. the index of the current bucket  
3. the next entry  
4. the expected mod count (for fail-fast)  
   
at every call to next(), the current is initialized as next and next is calculated (as well as the index)  
hasNext() returns false when next is null  
   
For iterating over key, value and entry objects, multiple implementations of HashIterator are provided  
  
[code]**private** **final** **class** ValueIterator **extends** HashIterator<V> {  
        **public** V next() {  
            **return** nextEntry().value ;  
        }  
    }  
   
    **private** **final** **class** KeyIterator **extends** HashIterator<K> {  
        **public** K next() {  
            **return** nextEntry().getKey();  
        }  
    }  
   
    **private** **final** **class** EntryIterator **extends** HashIterator<Map.Entry<K,V>> {  
        **public** Map.Entry<K,V> next() {  
            **return** nextEntry();  
        }  
    }[/code]  
  
  
rehashing :  
   
As a protection against poorly written hash functions, HashMap rehashes the hashcode of every key  
   
In HashMap the decision regarding which bucket a key should go to is made on the basis of this :  
  
[code] **static** **int** indexFor( **int** hash, **int** capacity) {  
        **return** hash & (capacity-1);  
   }[/code]  
  
Note that capacity is always a power of 2 (ie 1 followed by a sequence of zeroes in binary). So (capacity - 1) is a sequence of 1's in binary. So if the capacity is 2^n, then only the lower n bits of hash are useful and the upper bits beyond that are ignored  
   
Tests show that too many hash functions are not random enough in their lower bits, and that many hashmaps are not large enough to ever use the higher bits  
   
So, as a guard against poorly written hash functions, the hashcodes of the keys are rehashed  
  
dynamic resizing  
   
As the number of elements in the map grows, the hash chains will grow longer and retrieval time will increase. At some point, it makes sense to increase the number of buckets and rehash the values. Remember finding the correct bucket takes O(1) time while finding the correct entry in the bucket takes O(n) time. So it makes sense to have more number of buckets instead of having a few crowded buckets  
   
HashMap always doubles up the capacity during resizing. It creates a new bucket array of twice the size and copies data from old array to new array. Then the table variable is reinitialized  with the new array  
  
[code]    **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;  
    }[/code]  
  
During transfer from old array to new array, it iterates over the elements of the old array, for every Entry element computes the new bucket array index and sets the next pointer of thst element to the head of the bucket (ie adds elements to the head of the singly linked list). So, the order of the linked list elements are reversed  
   
Suppose original linkedlist was 1->2->3  
Lets assume after resizing, every element again goes into same bucket  
So, after first iteration, 1->null  
after second iteration, 2->1->null  
after third iteratuion, 3->2->1->null thus the link list gets reversed  
   
This creates a potential race condition during resizing. Suppose two threads parallely decide that the HashMap needs resizing and try to resize. This may lead to an infinite loop.  
   
Of course the user had no business using HashMap in a multithreaded context to begin with  
   
put operation  
   
The put operation performs the following steps :  
1. calculate hashcode for key  
2. rehash it. lets call the rehashed results as h  
3. calculate bucket index as h & (capacity -1)  
4. now iterate over the bucket and compare key with all existing keys using equals()  
5. if the key already exists, change the value of that Entry object  
6. else create a new Entry object and add to the head of the linked list  
7. increment mod count  
8. resize if necessary  
   
get operation  
   
The get operation performs the following steps :  
1. calculate hashcode for key  
2. rehash it. lets call the rehashed results as h  
3. calculate bucket index as h & (capacity -1)  
4. now iterate over the bucket and compare key with all existing keys using equals()  
5. if the key already exists return the corresponding value in the Entry object