COLLECTIONS – WHAT HAPPENS WITHIN

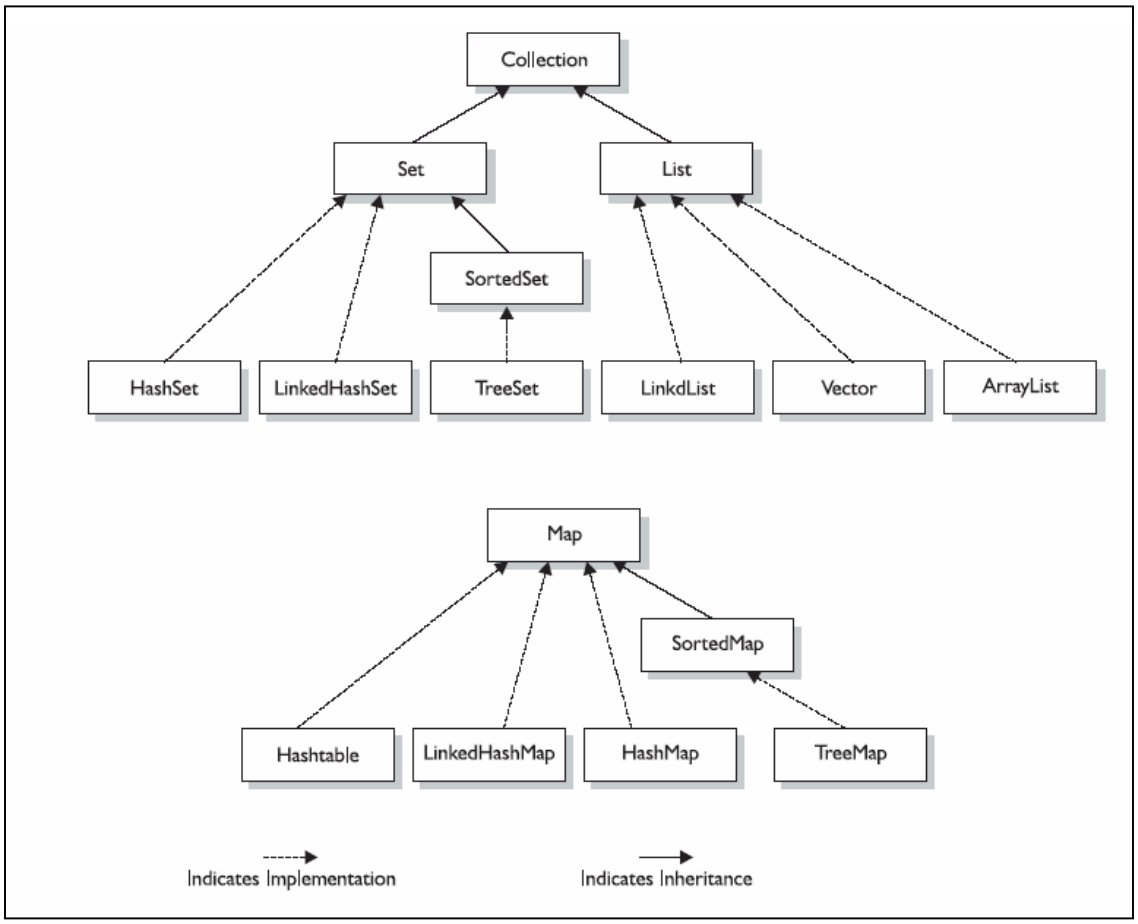
-Compiled by Sharad Ballepu

TABLE OF CONTENTS

1. Collection hierarchy 2
2. List 3
   2.1 ArrayList 3
   2.2 Vector 5
   2.3 Linked List 5
3. Map 13
   3.1 HashMap 13
   3.2 HashTable 20
   3.3 TreeMap 20
4. Set 21

Collections: Data structure that holds objects and provides various utility methods to manage the data.

Collections hierarchy



Three basic flavors of collections:

- Lists - Lists of things (classes that implement List)
- Sets - Unique things (classes that implement Set)
- Maps - Things with a unique ID (classes that implement Map)

The sub-flavors:

- Ordered - You can iterate through the collection in a specific order.
- Sorted - Collection is sorted in the natural order (alphabetical for Strings).
- Unordered
- Unsorted

► List extends Collection

In a List, every element has a index associated with it and we can add, delete, modify the objects in a List using the index.

Let's discuss about the 3 List implementations.

1) ArrayList
2) Vector
3) LinkedList

❖ ArrayList extends AbstractList implements List, Cloneable, java.io.Serializable:

- Resizable-array implementation of the List interface.
- Ordered collection.
- Should be considered when there is more of data retrieval than add/delete.
- Often used methods – add(), get(), remove(), set(), size().

Implementation details:

ArrayList uses arrays internally for data management. Whenever we invoke methods on the ArrayList object, it performs respective action on the array object to reflect the changes.

Constructing an ArryList: There are 3 ways in which one can construct an ArrayList i.e. ArrayList provides 3 constructors.

- Constructor that takes an integer argument – When the constructor with the integer argument is invoked, a single dimension array (which the ArrayList uses internally to manage the elements) is constructed with length equal to the value given in the constructor.
- Default constructor – When the default constructor is invoked, it in turn calls the constructor with integer arguments with the size as '10'.
- Constructor that takes a Collection as an argument. – Calculates the size of the Collection passed as constructs an array of length `(size*110)/100.`

Adding elements to the ArrayList: We can add an element to the end of the list, add an element at a particular index position, add a collection to the end of the list and add a collection at a particular index position.

Let is have a look at an example that adds an element to the end of the existing list.

```
ArrayList alObj = new ArrayList();
alObj.add("Element1");
```

What happens internally now when we invoke the add() method?

Add() calls `ensureCapacity(size + 1),` where size is the current size of the list, i.e. number of elements in the list. The `ensureCapacity(size + 1)` method compares the new capacity (the one we send as parameter i.e. size + 1) with the current capacity (remember that the initial capacity will be set to '10' when you construct an arraylist). Now, if the new capacity is more than the current capacity it will resize the array using the method

```
ExtendedSystem.resizeArray(newCapacity,elementData,0,size);

      newCapacity – desired length of the array
                    newCapacity = (oldCapacity * 3)/2 + 1;
      elementData – the array instance for copying the values
      size – the current length of the array.
```

That would mean that if the ArrayList was constructed using the default constructor, for the 1st 10 elements added the resizeArray will not be called. After that, depending upon the oldCapacity and newCapacity values the method would be invoked. After altering the size of the array, the new element will be placed at the end of the array.

On similar lines the other overloaded methods of add() works.

Retrieving elements from the ArrayList: There is only one way in which we can retrieve elements from an ArrayList i.e. throught the `get(int index)` method of the ArrayList. When this method is invoked it calls `RangeCheck(index)` which simply checks whether the given index is within the range of representing array. If yes, returns the element at that index else throws a `IndexOutOfBounds` exception.

Removing elements from the ArrayList: Uses `System.arraycopy(..)` method for deletion (the same concept is even used for adding elements to an arraylist at a specified position).

```
System.arraycopy(src,srcposition,dest,destposition,length);

  src – the source array.
  srcposition – index from where to copy the values from source.
  dest – the destination array.
  destposition - index from where to copy the values to dest.
  length – number of elements to copy from source to dest.
```

This method copies a number of elements from one array to another. Say for example we want to remove the element at the 5th position in the list which contains 10 elements – we give the source and destination as the same array and invoke the arraycopy() method with parameters such that the 5th element is deleted.

Src = elementData (array that holds the elements)
Srcposition = 6
Dest = elementData
Destposition = 5
Length = 4

Similarly, for adding elements at a particular index the same concept is employed.

We can also remove an element using the `remove(Object o)` method, which will remove the first occurrence of the object in the list by iterating through the list and checking for equality using the `obj1.equals(obj2)` method, returns false if the object is not found. **The equals method should be overridden for desired behavior.** If the equals method is not overridden it is taken from that of Object class (which only checks whether two references refer to the same object or not, but not check whether the content inside the objects are same or not).

- `Vector` extends AbstractList
  implements List, Cloneable,java.io.Serializable
  - Ordered collection
  - To be considered when thread safety is a concern.
  - Often used methods – add(), remove(), set(), get(), size().

Implementation details:

The Vector is also implemented on similar lines of ArrayList, but some of the methods in Vector are synchronized and Vector has more methods when compared to ArrayList.

- `LinkedList` extends AbstractSequentialList
  implements List, Cloneable,java.io.Serializable
  - Ordered collection.
  - Faster insertion/deletion and slower retrieval when compared to ArrayList.

Implementation details:

As the name suggests, it is a LinkedList implementation. The LinkedList class uses an inner class called Entry.

```
private static class Entry {
  Object element;
  Entry next;
  Entry previous;

  Entry(Object element, Entry next, Entry previous) {
      this.element = element;
      this.next = next;
      this.previous = previous;
  }
}
```

This class has 3 instance variables

```
element  – holds the object that is added to the LinkedList.
next     – reference to the next element in the list
           (which is of type Entry).
previous - reference to the previous element in the list
           (which is of type Entry).
```

That would mean that at any point of time we know the current object, its previous element and its next element.

Constructing a LinkedList:

There are only 2 constructors for a LinkedList. A no-arg constructor and the one which takes a collection as an argument.

Adding elements to the LinkedList: There are many overloaded methods provided for adding an element/collection to the LinkedList. Let us discuss how a simple `add(Object o)` method works (all other overloaded methods are implemented in the similar way).

When the method `add(Object o)` is invoked on the LinkedList instance, it in turn calls the method `addBefore(o, header).` We will discuss what is this 'header' and what is done in the addBefore() method.

For adding elements to a LinkedList we first create an instance of the LinkedList

```
LinkedList ll = new LinkedList();
```

Now, what happens in the LinkedList constructor? The LinkedList constructs looks like this.

```
public class LinkedList extends AbstractSequentialList
                    implements List, Cloneable, java.io.Serializable
{
      …..
      private transient Entry header = new Entry(null, null, null);
      public LinkedList()
      {
            header.next = header.previous = header;
      }
      …..
}
```

`header` is nothing but an instance of Entry (Entry class which we have seen earlier). Let us assume that every instance of the Entry class is a node. We will give a representation to this node.

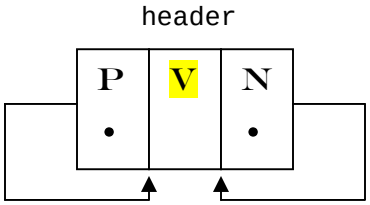


Fig. 1

P - reference to the preivious node. [type – Entry]
V - Value of the node(inserted object using add() or set()). [type – Object]
N - reference to the next node. [type – Entry]

Let us add an element to the LinkedList using the method `add(Object o).`

```
add("Element1");
```

This in turn calls the method `addBefore("Element1", header);` Now we know what `header` contains.

Let us have a look at what happens within addBefore(…).

```
1.   private Entry addBefore(Object o, Entry e)
2.   {
3.     Entry newEntry = new Entry(o, e, e.previous);
4.     newEntry.previous.next = newEntry;
5.     newEntry.next.previous = newEntry;
6.     size++;
7.     modCount++;
8.     return newEntry;
9.   }
```

At line3. a new instance of type Entry is created. This will represent the object we have added. The 1st parameter is the value(added object), 2nd parameter is the reference to the next element and the 3rd parameter is the reference to the previous element in the constructor. How does the node structure look like now?

```
Entry newEntry = new Entry(o, e, e.previous);
```

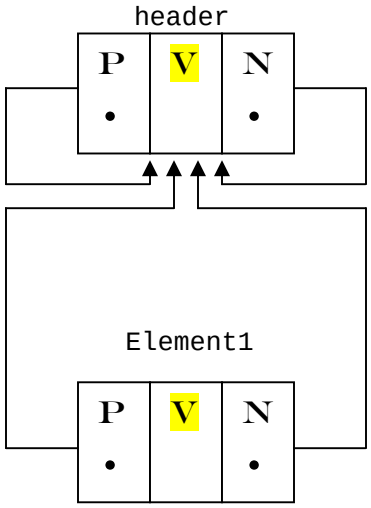


Fig. 2

As represented in fig.2 the 1st node is the `header` and the 2nd node is the new instance of Entry `Element1`.

In the Entry constructor 'e' is nothing but `header.` We know what happens in the Entry constructor (see page 4). After the constructor runs, we have:

```
Element1.element   = o
                   = "Element1"

Element1.next      = e
                   = header

Element1.previous  = e.previous
                   = header.previous
                   = header.
```

What happens at lines 4 and 5?

```
newEntry.previous.next = newEntry;
Element1.previous.next = Element1 (as newEntry == Element1)
header.next            = Element1 (as Element1.previous = header)

newEntry.next.previous = newEntry
Element1.next.previous = Element1 (as newEntry == Element1)
header.previous        = Element1 (as Element1.next == header)
```

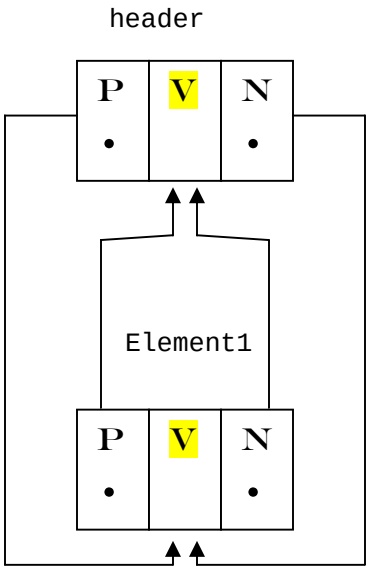


Fig. 3

This is how the node structure appears after adding an element to the LinkedList. `header` is the root element and is always null. The `next` element for `header` is `Element1` and the `previous` element for `Element1` is `header`.

Let us see what happens when we add another element to the list, which gives a clear idea about the logic implemented.

Assume we added another element to the list `add("Element2");`
Let us have a look at the node structure after line3 (page 6).

```
Entry newEntry = new Entry(o, e, e.previous); (e is nothing but header)
```

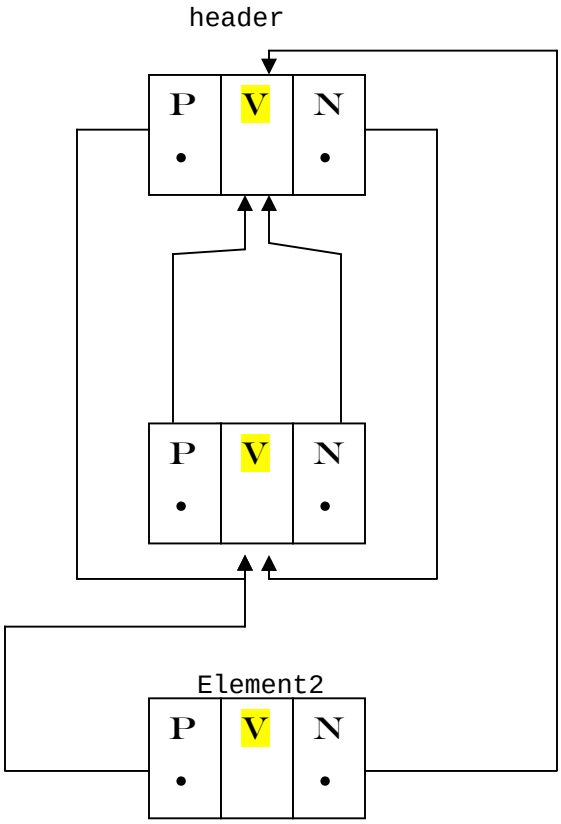


Fig. 4

```
Element2.element  = o = "Element2"
Element2.next = e = header
Element2.previous  = e.previous
                   = header.previous
                   = Element1.
```

Now, the the new element's (`Element2`) previous element will be `Element1` and the new element's next element will be `header`

Looking at fig. 4 above we can figure out that there are 2 more things to be done:

1) The next for `Element1` should be `Element2`
2) The previous for `header` should be `Element2`.

This is achieved after lines 4 and 5(page 6) executes.

```
newEntry.previous.next = newEntry;
Element2.previous.next = Element2 (as newEntry == Element2)
Element1.next          = Element2 (as Element2.previous = Element1)

newEntry.next.previous = newEntry
Element2.next.previous = Element2 (as newEntry == Element2)
header.previous        = Element2 (as Element2.next == header)
```

Finally, how does it look?

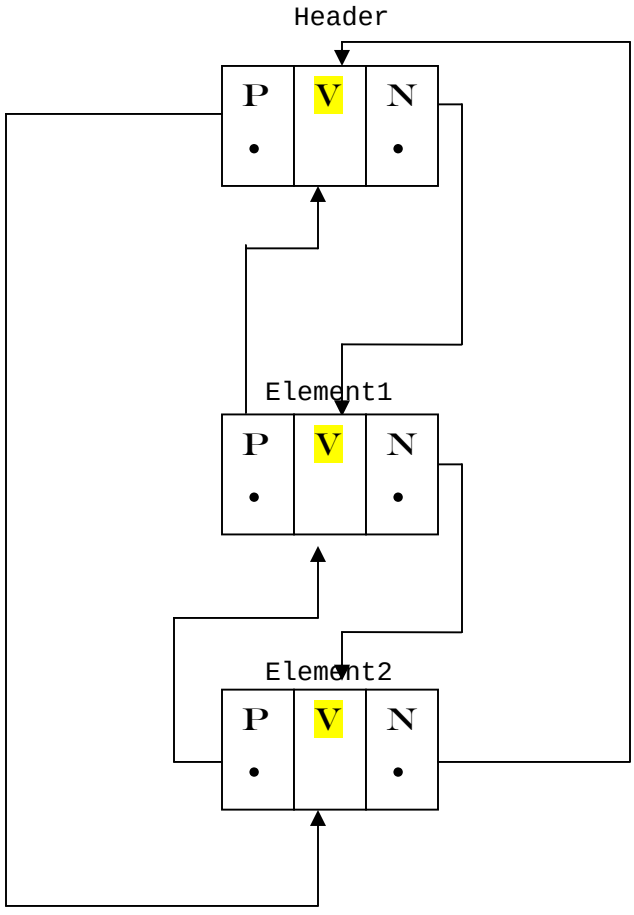


Fig. 5

As the elements are added to the LinkedList, the next always holds a reference to the header and the previous always holds a reference to the last element of the list.

Similarly when we add elements to the LinkedList at a particular index using the overloaded add() method, it invokes the same `addBefore()` method, but passes the current element at the index(which is passed as parameter to the add() method) instead of `header.`

The element at a particular index in the list is retrieved by calling `element.next` index number of times.

```
for (int i = 0; i <= index; i++)
   e = e.next;
```

In the same fashion, the element at a particular index is retrieved and replaced with the new element when we call `set(int index,Object element)` on the LinkedList.

Retrieving elements from the LinkedList: There are 3 methods provided for retrieving elements from a LinkeList.

`get (int index)` – retrieves the element at index as explained above for add() and set().

`getFirst()` – As we know the first element is always `header` (which is null and not the first element added by us). It returns `header.next.element`.

`getLast()` – The previous element of the header will always be the last element of the list(see fig. 5). So, this method returns `header.previous.element.`

Removing elements from the LnkedList: All the overloaded methods for get() are also applicable to remove(). We `have remove(int index)`, `removeFirst(), removeLast()` which is implemented in the same way as its corresponding get() methods. We have one more overloaded method for remove() i.e.

`remove(Object o)` – traverses the list from header, if `o.equals(e.element)` and returns a boolean. Override the equals() method for desired functionality.

►Map:

Map is a collection that holds key/value pairs. A unique key is mapped to a specific value, where both the 'key' and 'value' in are objects.

Some of the Map implementations are:

1) HashMap
2) Hashtable
3) LinkedHashMap
4) TreeMap

❖ HashMap extends AbstractMap,
implements Map, Cloneable, java.io.Serializable

- An unsorted, unordered Map.
- Allows a *null* key and multiple *null* values.
- Methods are not synchronized.
- Often used methods: get(), put(), size(), values(), containsKey()…

Implementation details:

HashMap stores elements in an array of type Entry

```
private transient Entry table[];
```

`Entry` is a inner class that holds the hash, key, value and the next element's information.

```
private static class Entry implements Map.Entry
{
    int hash;
    Object key;
    Object value;
    Entry next;

    Entry(int hash, Object key, Object value, Entry next) {
        this.hash = hash;
        this.key = key;
        this.value = value;
        this.next = next;
}
```

```
hash – hashCode value of the object
key    - key inserted using hashMapObj.put(key,value) method
value – value associated with the key
next – reference to the next element in the HashMap.
```

What is hashCode?

A hashCode is nothing but a simple integer value that determines at which index position in `table[]` the elements goes into the HashMap. We can override the hashCode() method which should return an integer value. The hashCode contract says that when two objects are equal (i.e. obj1.equals(obj2) == true) their hashCodes must also be equal (but when obj1.equals(obj2) == false, they can still have the same hashCode).

Generally, we override the hashCode() method to group similar objects together. If we are not sure how to implement the hashCode() we can leave it for the default implementation.

Constructing a HashMap: A hashMap has 4 overloaded constructors.

`HashMap(int initialCapacity, float loadFactor)` – creates a new array 'table' of type `Entry` and of size `initialCapacity`. The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased.

A threshold is calculated using the formula
```
threshold = (int)(initialCapacity * loadFactor);
```

`HashMap(int initialCappacity)` – in turn calls the 1st overloaded method passing the `loadFactor` value as 0.75

`HashMap()` – calls the 1st overloaded method with `initialCapacity` 11, and `loadFactor` 0.75

`HashMap(Map t)` – calls the 1st overloaded method and puts all elements of Map t in the newly created one.

Adding elements to a HashMap:

When an element is added to the HashMap, the following steps are executed.

1) Retrieves the hashCode value of the inserted object.

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

   If the `hashCode()` method is overridden in the object's class it is taken from there, else the default implementation from the Object class is taken. Unless you need to give some special implementation it is advisable not to override the `hashCode()` method. Whenever `hashCode()` is overridden the hashCode contract has to be followed.

2) Computes the index at which the new element (the key and value which we pass to the put() method) will be placed in the table[].

```
index = (hash & 0x7FFFFFFF) % tab.length;
```

The `tab` here is nothing but table[] (which holds all elements of type Entry). So, `tab.length` is nothing but the table length and a mod operation on tab.length will always give a number less than tab.length.

3) Inserts the element at the correct computed index.

There is quite a bit of logic that goes behind step3. It doesn't simply add the element at the computed index. Let us look at a small example that adds a few elements to the HashMap and understand what happens behind the scenes.

Example: We will add a few Strings to the HashMap. We all know that the String class overrides both the hashCode() and equals() methods.

```
HashMap hm = new HashMap(); - creates a new HashMap instance.
                            - Calls the overloaded constructor giving
                            - the initialCapacity as 11 and loadFactor
                              0.75.
                            - Initializes the array which is used to
                              hold all the entries – table[].
                            - calculates the threshold value.
```

The table[] array will look something like this now

```
Entry table[] = {null, null, null, null, null, null….};
```

Now that the HashMap is ready, let us start adding elements to it and visualize how the HashMap class handles it.

```
hm.put("1", "element1"); - Added an element with key – "1" and
                           value – "element1".
```

Within the HashMap class:

1) As the key of the element to be added is not null, the hash value and the index is calculated. Let us assume that the calculated index was computed to "4". That would mean that this element will be added at index 4 in the table[] array.
2) A check is made whether an element with the given key (i.e. "1" in this case) is already present in the Map. (We know that there are no elements in the HashMap so we will ignore this at this point of time).

3) Makes a check on the threshold. If the threshold is exceeded then calls the function `rehash()` which will create a new array with more capacity, calculates new indexes for all existing elements (so that the elements are equally distributed throughout the table) in the Map and adds the elements to the table at the calculated index.

```
if (count >= threshold) {
    // Rehash the table if the threshold is exceeded
    rehash();

    tab = table;
    index = (hash & 0x7FFFFFFF) % tab.length;
}
count is incremented by 1 each time a new element is added to
the table.
```

The new capacity for table will be: `newCapacity = oldCapacity * 2 + 1.` and the new array is created using the method `ExtendedSystem.newArray()`.

We also know that the `rehash()` function will not be called at this point of time as it is the 1st element of the Map.

So, simply a new element is added to the table[] (remember that table is a reference to a single dimension array of Objects of type Entry) and the count is incremented by 1.

```
Entry e = new Entry(hash, key, value, tab[index]);
          // here tab[index] is null as there were no elements
          // at that index before.
tab[index] = e;
count++;
```

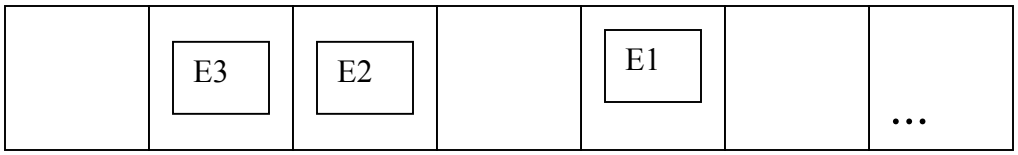
In the same fashion let us add 2 more new elements to the Map.

```
hm.put("2", "element2");
hm.put("3", "element3");
```

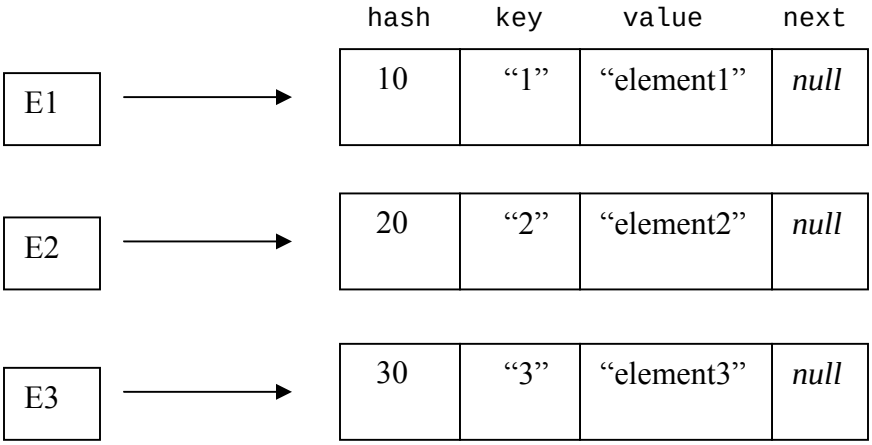
The `hashCode()` of any of these three keys will not be the same (because equals doesn't return true for any two comparisons), and assume that the indexes computed for these 3 elements are different.

Let us visualize the array table[] and its elements.

table:



Index 0 1 2 3 4 5



Things were very simple till now. We had 3 elements with different hashCode values and different indexes which went into 3 different places in the table[]. But what if the hashCode for any two elements are same or the computed index for 2 elements is the same? With a good `hashCode()` implementation, two similar/related objects will return the same hashCode thus same index in the table. And these two elements will then share the same index in the table[] – but how can we have two objects at the same index?

Lets consider this case…
Add another element to the HashMap

```
hm.put("5", "element5");
```

Assume that there is some relation between element3 and element5 and the `hashCode()` returns the same value for both these objects thus same index. i.e. index 1 (relationship can be that both are prime numbers).

We have seen how a new element is added to the table[].

```
Entry e = new Entry(hash, key, value, tab[index]);
tab[index] = e;
count++;
```

So when we say `hm.put("5", "element5");`, the above piece of code is executed. That would imply that this new element is added at index 1 in table[]. So, what about the previous element that was present at this index? The element should not be replaced with the old one as they are two different elements with different key/value pair.

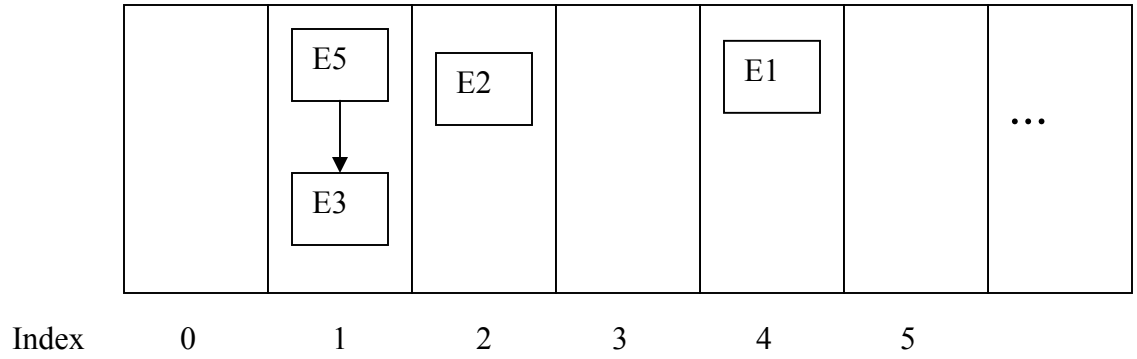
```
Entry e = new Entry(hash, key, value, tab[index]);
```

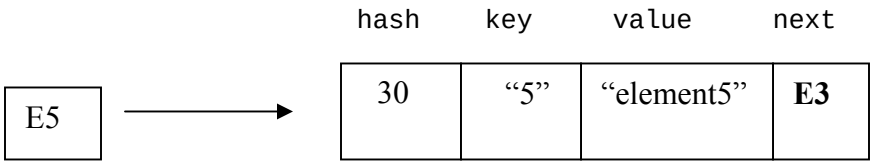
After having a look at the 4th parameter of the constructor we can figure out what's going on. The 4th parameter holds the reference of the object that is currently sitting at index 1 in the table.

So, `element5.next = element3;`

How does the table look now?

table:





Thus elements are managed in the form of LinkedList within the indexes.

What happens when we give the same `key` to the HashMap with a different `value`? Say for example I want to have another entry in the HashMap with key "1" and value "element4". ---- this is not possible with HashMap (for that matter any Map implementation). A "key" to the map is always a unique value and if we try to add something like the above the original value will be replaced with the new value.

```
if (key != null)
{
    hash = key.hashCode();
    index = (hash & 0x7FFFFFFF) % tab.length;
    for (Entry e = tab[index] ; e != null ; e = e.next)
    {
        if ((e.hash == hash) && key.equals(e.key))
       {
            Object old = e.value;
            e.value = value;
            return old;
        }
    }
```

As we can see from the above code snippet, it checks if there are any existing elements with the same `key` and replaces the old `value` with the new one.

Retrieving elements from a HashMap: With the hashing technique employed to put elements in the HashMap, it becomes very much easier to retrieve elements from it.

The method to retrieve an element from a HashMap is `get(Object o)`, where the o is the nothing but the `key` for whose `value` we are looking. This method returns an Object – the `value` associated with the given `key`, returns *null* if there is no mapping for the key.

`hm.get("1"); - will retrieve the value associated with key "1".`

First the hashCode for the key is calculated followed by the index. For sure, the element will be present at the computed index (if there is an element with that key in the table). A check is made for the key for all elements at that index location. (next element is retrieved using `element.next`).

Removing elements from a HashMap: Elements are removed from the HashMap in the same fashion as in LinkedList.

❖ Hashtable extends Dictionary,
implements Map, Cloneable,java.io.Serializable

- Map implementation.
- Un-ordered collection, un-sorted collection.
- Should be considered when security is important.
- Often used methods – put(), get(), remove(), values(),keySet().

Implementation details:

A Hashtable is implemented in the same fashion as HashMap.

Things in Hashtable that is not a feature of HashMap:

- ✓ has synchronized methods.
- ✓ has more utility methods when compared to HashMap.
- ✓ doesn't allow *null* keys or values.

❖ TreeMap extends AbstractMap
implements SortedMap, Cloneable, java.io.Serializable

- Map implementation.
- Sorted by keys in the natural order.

The elements in a TreeMap are sorted by the keys.
As with every Map, only objects can be stored as keys and values in a TreeMap.
A TreeMap allows null values.
All keys that go into the TreeMap should implement the Comparable interface.
The only method in the Comparable interface is the `compareTo(Object o);` Your class should override this method which returns an int

Negative value – this object is less than the specified object.
Zero - this object is equal to the specified object.
Positive value - this object is greater than the specified object.

String class overrides the compareTo() method.
As we have seen that most of the collection classes have an inner class Entry.

►Set extends Collection

A Set is a collection where all the values are unique.
A Set is similar to List (which holds a collection objects), with the only difference that in a Set all the elements are unique, i.e. there cannot be 2 elements with the same value in a Set.
All Set implementations use Map internally.

HashSet – uses HashMap internally.
LinkedhashSet – uses LinkedHashMap
TreeSet - uses TreeMap

The objects added to the all Set implementations are stored as keys to the Map(the value is a constant).

TABLE 7-2 Collection Interface Concrete Implementation Classes

Class	Map	Set	List	Ordered	Sorted
HashMap	X			No	No
Hashtable	X			No	No
TreeMap	X			Sorted	By *natural order* or custom comparison rules
LinkedHashMap	X			By insertion order or last access order	No
HashSet		X		No	No
TreeSet		X		Sorted	By *natural order* or custom comparison rules
LinkedHashSet		X		By insertion order or last access order	No
ArrayList			X	By index	No
Vector			X	By index	No
LinkedList			X	By index	No

* All implementations as per jdk1.3
* References - Sun Certified Programmer & Developer for Java2 – Kathy Sierra, Bert Bates.
(Collection hierarchy diagram, last diagram and a few definitions taken from this book).