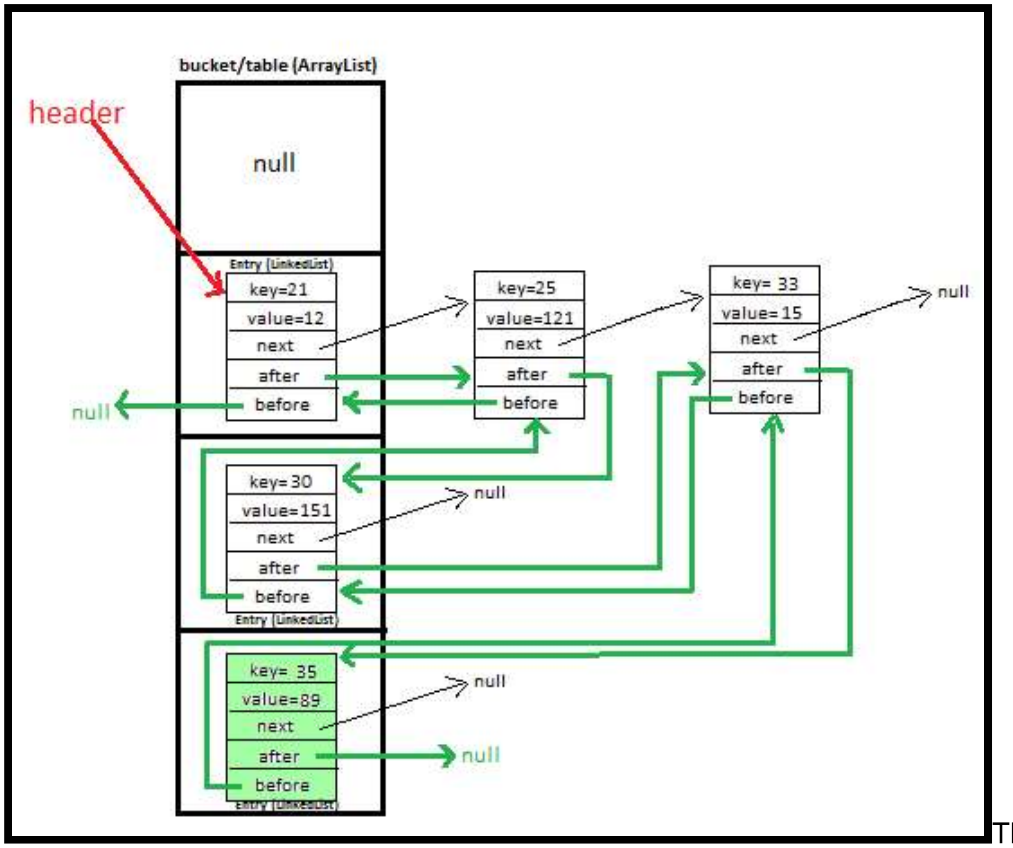
LinkedHashMap Custom implementation in java - How LinkedHashMap works internally with diagrams and full program

Contents of page :

1) Custom LinkedHashMap >



This is very **important and trending** topic. In this post i will be explaining **LinkedHashMap** custom implementation with diagrams which will help you in **visualizing** the LinkedHashMap implementation.

I will be explaining how we will **put** and **get** key-value pair in HashMap by overriding-
>**equals** method - helps in checking equality of entry objects.
>**hashCode** method - helps in finding bucket's index on which data will be stored.

We will maintain **bucket (**ArrayList**)** which will store **Entry (**LinkedList**).**

Most salient feature of **LinkedHashMap** is that it **maintains insertion order** of key-value pairs. We will maintain doubly Linked List for doing so.

While our HashMap didn't maintained insertion order.

2) Entry<K,V>

We store key-value pair by using `Entry<K,V>`

By using, `Entry<K,V> before, after` - we keep track of newly added entry in LinkedHashMap, which helps us in **maintaining insertion order**.

Entry contains

- K **key,**
- V **value,**
- `Entry<K,V>` **next** (i.e. next entry on that location of bucket),
- `Entry<K,V>` **before** and
- `Entry<K,V>` **after**

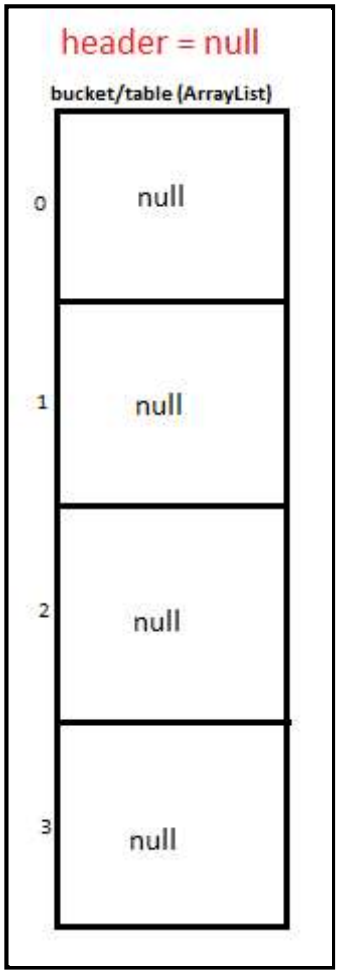
```
        static class Entry<K, V> {
         K key;
         V value;
         Entry<K,V> next;
         Entry<K,V> before, after ;

         public Entry(K key, V value, Entry<K,V> next){
             this.key = key;
             this.value = value;
             this.next = next;
         }
    }
```

*3) Putting **5 key-value pairs in own/custom LinkedHashMap (step-by-step)>***

I will explain you the whole concept of **LinkedHashMap** by putting **5 key-value pairs in HashMap.**

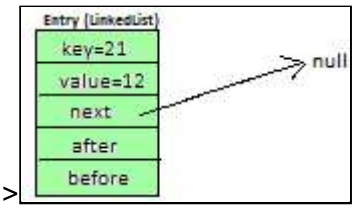
Initially, we have bucket of **capacity=4.** (all indexes of bucket i.e. 0,1,2,3 are pointing to null)



Let's put first key-value pair in LinkedHashMap-

Key=21, value=12

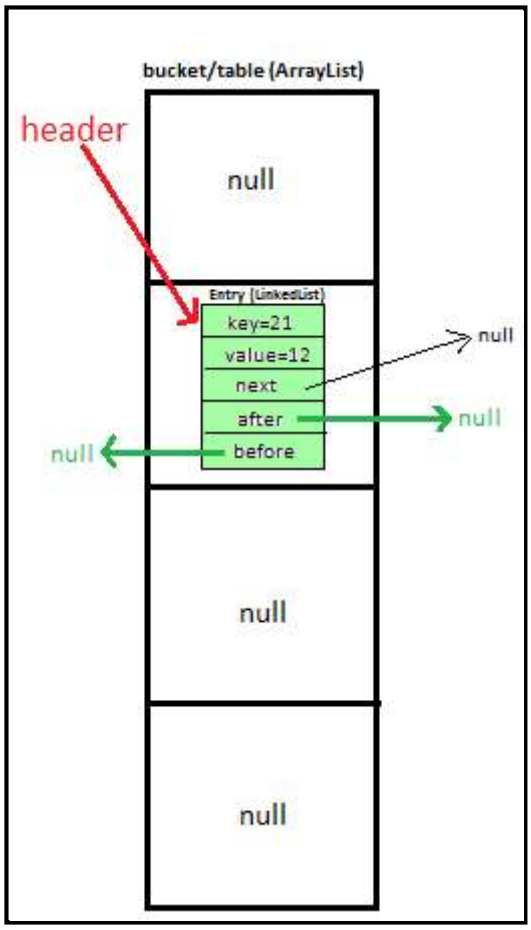
newEntry Object will be formed like this >



We will calculate hash by using our **hash(K key)** method **-** in this case it returns **key/capacity= 21%4= 1.**
So, **1** will be the **index of bucket** on which **newEntry object** will be stored**.**
We will go to **1st** index as it is pointing to null we will **put our newEntry object there.**

Additionally, for maintaining insertion order-
Update **header**, it will start pointing to **newEntry object**

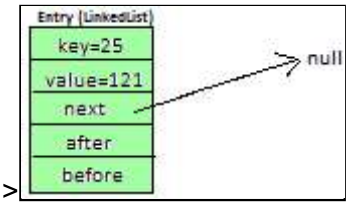
At completion of this step, our HashMap will look like this-



Let's put second key-value pair in LinkedHashMap-

Key=25, value=121

newEntry Object will be formed like this >



We will calculate hash by using our **hash(K key)** method - in this case it returns **key/capacity= 25%4= 1.**

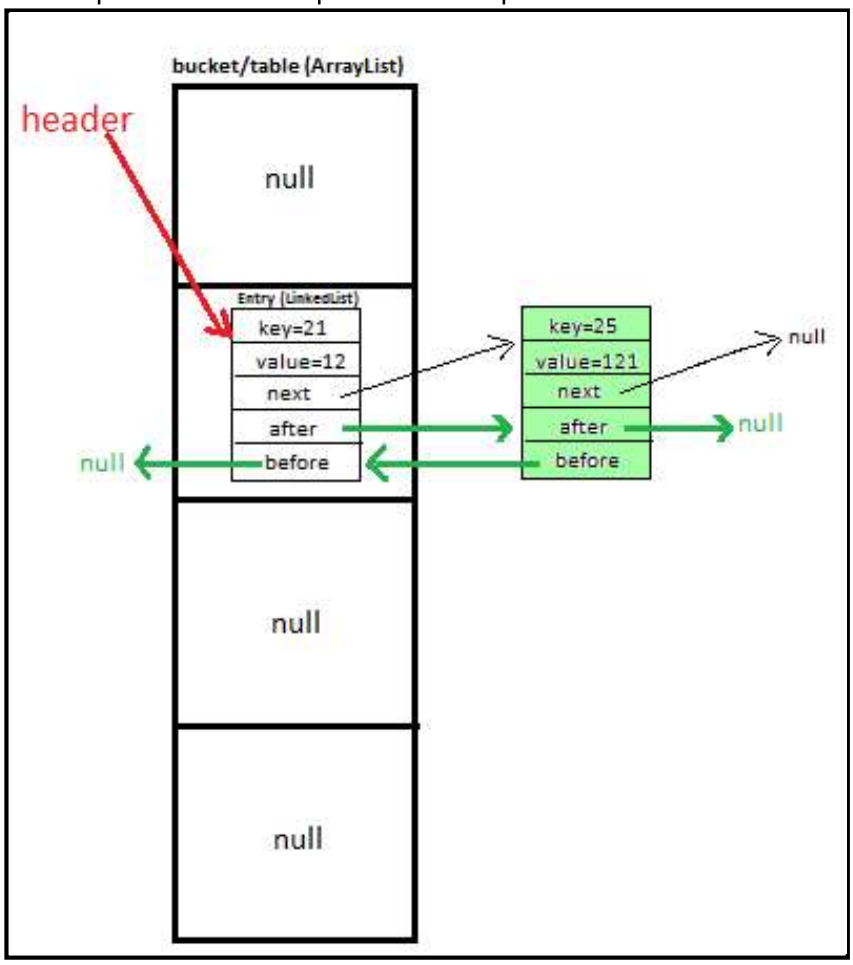
So, 1 will be the **index of bucket** on which **newEntry object** will be stored**.**

We will go to **1st** index, it contains **entry with key=21**, we will compare two keys(i.e. **compare 21 with 25** by using **equals method**), as **two keys are different** we check whether entry with key=21's **next is null or not**, **if next is null** we will **put** our **newEntry object** on **next.**

Additionally, for maintaining insertion order-

Update **header.after**, it will start pointing to **newEntry object** (i.e make Entry with key=21's after point to **newEntry object**], and also make **newEntry object's** before point to header (Entry with key=21')

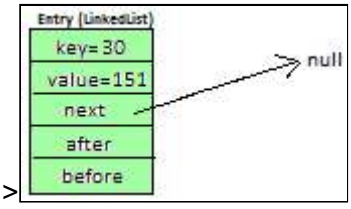
At completion of this step our HashMap will look like this-



Let's put third key-value pair in HashMap-

Key=30, value=151

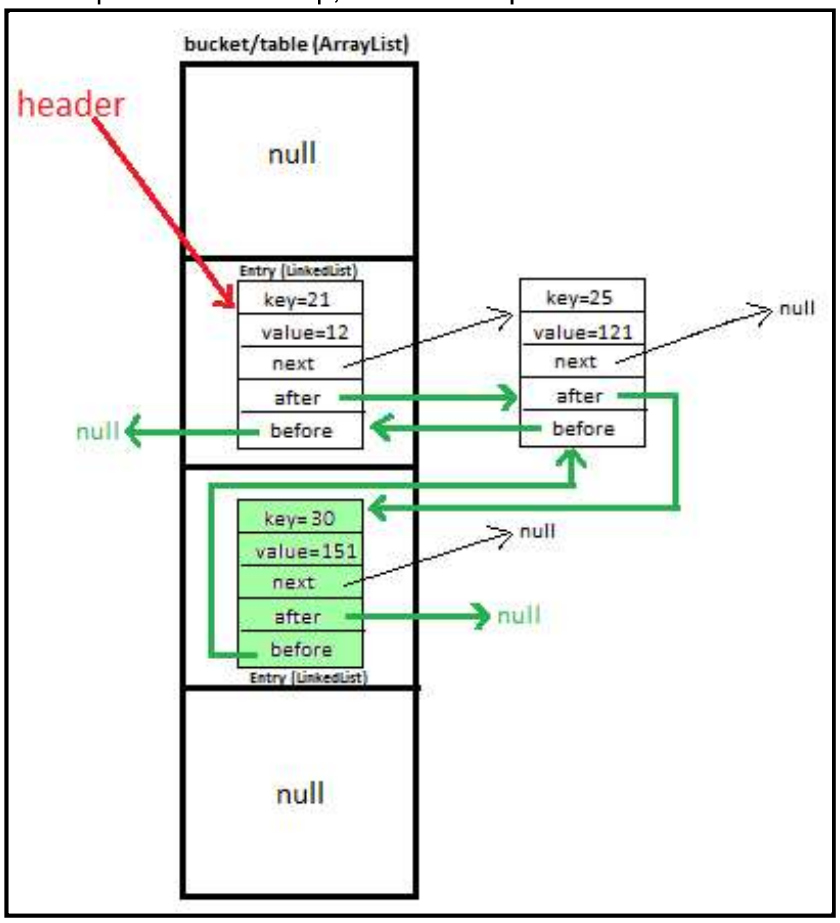
newEntry Object will be formed like this >



We will calculate hash by using our **hash(K key)** method **-** in this case it returns **key/capacity= 30%4= 2.**
So, **2** will be the **index of bucket** on which **newEntry object** will be stored**.**
We will go to **2nd** index as it is pointing to null we will **put our newEntry object there.**

Additionally, for maintaining insertion order-
Update doubly linked list 's **after and before.**

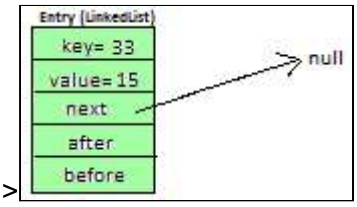
At completion of this step, our HashMap will look like this-



Let's put fourth key-value pair in LinkedHashMap-

Key=33, value=15

Entry Object will be formed like this >



We will calculate hash by using our **hash(K key)** method - in this case it returns
key/capacity= 33%4= 1,
So, 1 will be the **index of bucket** on which **newEntry object** will be stored.

We will go to **1st** index -

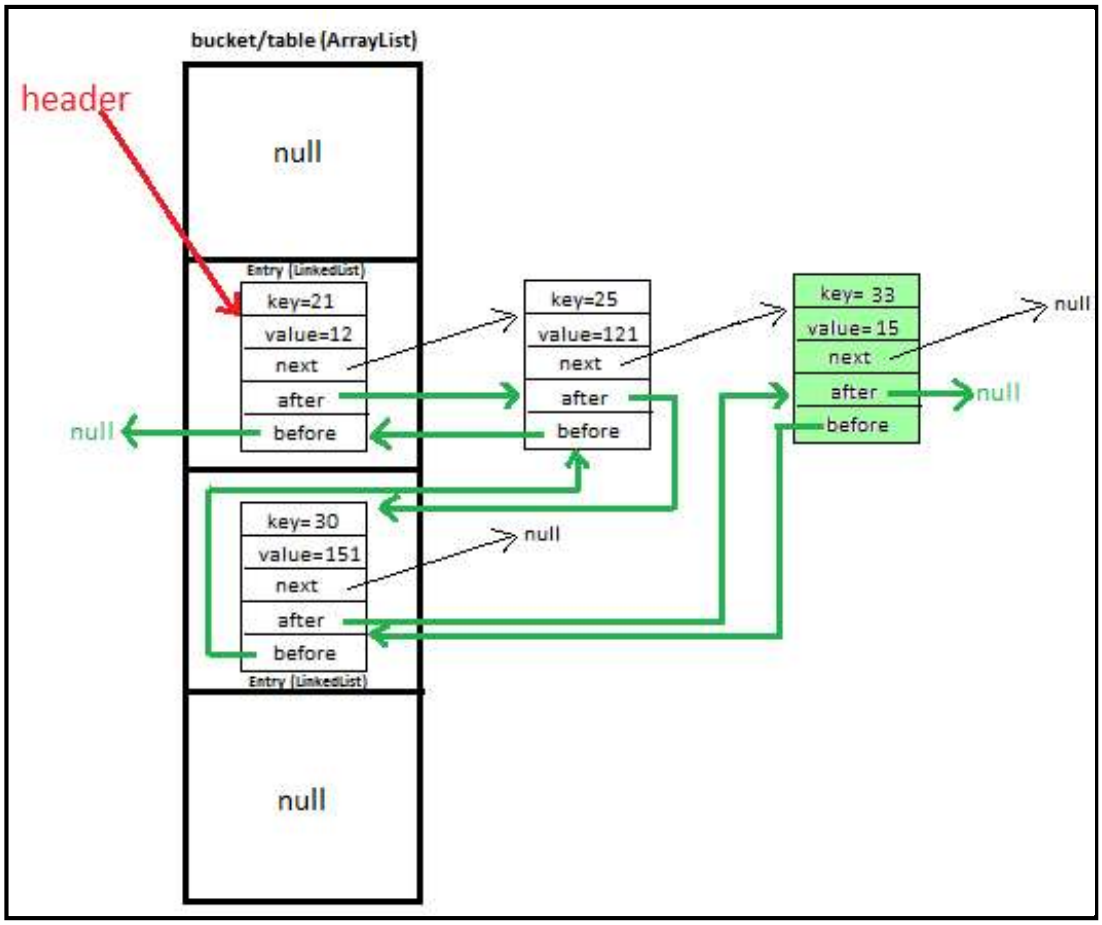
>it contains **entry with key=21**, we will **compare** two keys (i.e. **compare 21 with 33** by using **equals method**, as **two keys are different,** proceed to next of **entry with key=21 (**proceed only if **next is not null).**

>now, next contains **entry with key=25**, we will **compare** two keys (i.e. **compare 25 with 33** by using **equals method**, as **two keys are different,** now **next of entry with key=25** is pointing to **null** so we won't proceed **further,** we will **put our newEntry object on next.**

Additionally, for maintaining insertion order-

Update doubly linked list's **after and before** (for maintaining insertion order)

At completion of this step our HashMap will look like this-

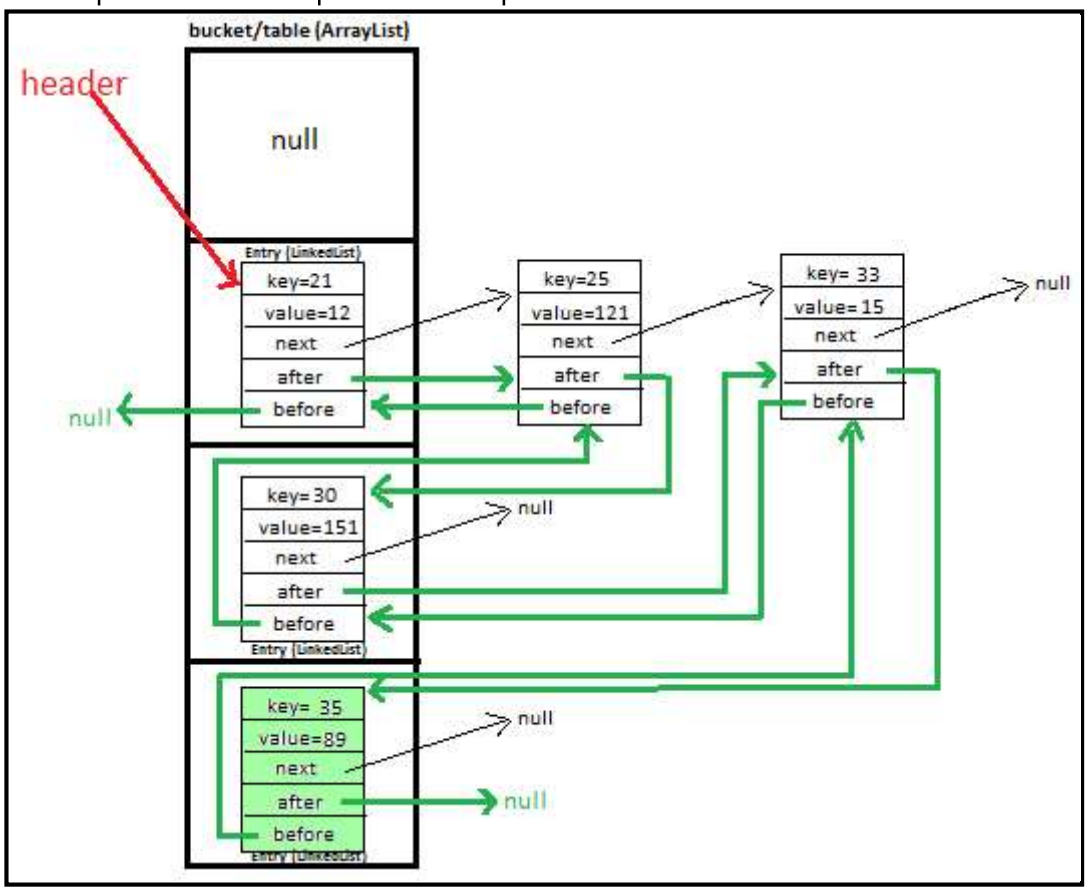


Let's put fifth key-value pair in LinkedHashMap-

Key=35, value=89

Repeat above mentioned steps.

At completion of this step our HashMap will look like this-



Must read: Set Custom implementation.

4) Methods used in custom LinkedHashMap >

public void **put**(K newKey, V data)	-Method allows you put key-value pair in HashMap -If the map already contains a mapping for the key, the old value is replaced. -provide complete functionality how to override equals method. -provide complete functionality how to override hashCode method.
public V **get**(K key)	Method returns value corresponding to key.
public boolean **remove**(K deleteKey)	Method removes key-value pair from **LinkedHashMapCustom.**
public void **display**()	-Method displays all key-value pairs present in **LinkedHashMapCustom.**, **-insertion order is guaranteed.**
private int **hash**(K key)	-Method implements hashing functionality, which helps in

	finding the appropriate bucket location to store our data. -This is very important method, as performance of **LinkedHashMapCustom** is very much dependent on this method's implementation.
private void **maintainOrderAfterInsert**(Entry<K, V> newEntry)	Methods helps in maintaining insertion order after insertion of key-value pair.
private void **maintainOrderAfterDeletion**(Entry<K, V> deleteEntry)	Methods helps in maintaining insertion order after deletion of key-value pair.

For more Refer : [LinkedHashMap Custom implementation - put, get, remove Employee object]

5) Full Program/SourceCode for implementing custom LinkedHashMap>

```
package com.ankit;

 /**
 * @author AnkitMittal, JavaMadeSoEasy.com
 * Copyright (c), AnkitMittal . All Contents are copyrighted and must not be
 * reproduced in any form.
 * This class provides custom implementation of LinkedHashMap(without using java api's)-
 * which allows us to store data in key-value pair form.
 * It maintains insertion order, uses DoublyLinkedList for doing so.
 * If key which already exists is added again, its value is overridden but
 * insertion order does not change,
 * BUT, if key-value pair is removed and value is again added than insertion order
 * changes(which is quite natural behavior).
 * @param <K>
 * @param <V>
 */
class LinkedHashMapCustom<K, V> {

    private Entry<K,V>[] table;   //Array of Entry.
    private int capacity= 4;  //Initial capacity of HashMap
    private Entry<K,V> header; //head of the doubly linked list.
    private Entry<K,V> last; //last of the doubly linked list.

    /*
     * before and after are used for maintaining insertion order.
     */
    static class Entry<K, V> {
        K key;
        V value;
        Entry<K,V> next;
        Entry<K,V> before,after;

        public Entry(K key, V value, Entry<K,V> next){
            this.key = key;
```

```
            this.value = value;
            this.next = next;
        }
    }


@SuppressWarnings("unchecked")
public LinkedHashMapCustom(){
   table = new Entry[capacity];
}



/**
 * Method allows you put key-value pair in LinkedHashMapCustom.
 * If the map already contains a mapping for the key, the old value is replaced.
 * Note: method does not allows you to put null key thought it allows null values.
 * Implementation allows you to put custom objects as a key as well.
 * Key Features: implementation provides you with following features:-
 *     >provide complete functionality how to override equals method.
 *  >provide complete functionality how to override hashCode method.
 * @param newKey
 * @param data
 */
public void put(K newKey, V data){
  if(newKey==null)
      return;    //does not allow to store null.

  int hash=hash(newKey);

  Entry<K,V> newEntry = new Entry<K,V>(newKey, data, null);
  maintainOrderAfterInsert(newEntry);
   if(table[hash] == null){
    table[hash] = newEntry;
   }else{
      Entry<K,V> previous = null;
      Entry<K,V> current = table[hash];
      while(current != null){ //we have reached last entry of bucket.
      if(current.key.equals(newKey)){
          if(previous==null){  //node has to be insert on first of bucket.
                newEntry.next=current.next;
                table[hash]=newEntry;
                return;
          }
          else{
              newEntry.next=current.next;
              previous.next=newEntry;
              return;
          }
      }
      previous=current;
        current = current.next;
    }
    previous.next = newEntry;
   }
}


/**
 * below method helps us in ensuring insertion order of LinkedHashMapCustom
 * after new key-value pair is added.
```

```
     */
    private void maintainOrderAfterInsert(Entry<K, V> newEntry) {

       if(header==null){
           header=newEntry;
           last=newEntry;
           return;
       }

       if(header.key.equals(newEntry.key)){
           deleteFirst();
           insertFirst(newEntry);
           return;
       }

       if(last.key.equals(newEntry.key)){
           deleteLast();
           insertLast(newEntry);
           return;
       }

       Entry<K, V> beforeDeleteEntry=    deleteSpecificEntry(newEntry);
       if(beforeDeleteEntry==null){
           insertLast(newEntry);
       }
       else{
           insertAfter(beforeDeleteEntry,newEntry);
       }


    }

    /**
     * below method helps us in ensuring insertion order of LinkedHashMapCustom,
     * after deletion of key-value pair.
     */
    private void maintainOrderAfterDeletion(Entry<K, V> deleteEntry) {

       if(header.key.equals(deleteEntry.key)){
           deleteFirst();
           return;
       }

       if(last.key.equals(deleteEntry.key)){
           deleteLast();
           return;
       }

       deleteSpecificEntry(deleteEntry);

    }

    /**
     * returns entry after which new entry must be added.
     */
    private void insertAfter(Entry<K, V> beforeDeleteEntry, Entry<K, V> newEntry) {
       Entry<K, V> current=header;
           while(current!=beforeDeleteEntry){
                  current=current.after;  //move to next node.
           }
```

```
        newEntry.after=beforeDeleteEntry.after;
        beforeDeleteEntry.after.before=newEntry;
        newEntry.before=beforeDeleteEntry;
        beforeDeleteEntry.after=newEntry;

    }


    /**
     * deletes entry from first.
     */
    private void deleteFirst(){

       if(header==last){ //only one entry found.
                header=last=null;
                return;
            }
            header=header.after;
            header.before=null;

    }

    /**
     * inserts entry at first.
     */
    private void insertFirst(Entry<K, V> newEntry){

            if(header==null){ //no entry found
                    header=newEntry;
                    last=newEntry;
                    return;
            }

            newEntry.after=header;
            header.before=newEntry;
            header=newEntry;

    }

    /**
     * inserts entry at last.
     */
    private void insertLast(Entry<K, V> newEntry){

            if(header==null){
                    header=newEntry;
                    last=newEntry;
                    return;
            }
            last.after=newEntry;
            newEntry.before=last;
            last=newEntry;

    }

    /**
     * deletes entry from last.
     */
    private void deleteLast(){

            if(header==last){
```

```
                header=last=null;
                return;
        }

        last=last.before;
        last.after=null;
}


/**
 * deletes specific entry and returns before entry.
 */
private Entry<K, V> deleteSpecificEntry(Entry<K, V> newEntry){

        Entry<K, V> current=header;
        while(!current.key.equals(newEntry.key)){
                if(current.after==null){   //entry not found
                        return null;
                }
                current=current.after;  //move to next node.
        }

        Entry<K, V> beforeDeleteEntry=current.before;
        current.before.after=current.after;
        current.after.before=current.before;  //entry deleted
        return beforeDeleteEntry;
}


/**
 * Method returns value corresponding to key.
 * @param key
 */
 public V get(K key){
    int hash = hash(key);
    if(table[hash] == null){
     return null;
    }else{
     Entry<K,V> temp = table[hash];
     while(temp!= null){
         if(temp.key.equals(key))
             return temp.value;
         temp = temp.next; //return value corresponding to key.
     }
     return null;   //returns null if key is not found.
    }
}


/**
 * Method removes key-value pair from HashMapCustom.
 * @param key
 */
 public boolean remove(K deleteKey){

   int hash=hash(deleteKey);

  if(table[hash] == null){
        return false;
  }else{
```

```
        Entry<K,V> previous = null;
        Entry<K,V> current = table[hash];

        while(current != null){ //we have reached last entry node of bucket.
           if(current.key.equals(deleteKey)){
               maintainOrderAfterDeletion(current);
               if(previous==null){  //delete first entry node.
                     table[hash]=table[hash].next;
                     return true;
               }
               else{
                     previous.next=current.next;
                   return true;
               }
           }
           previous=current;
             current = current.next;
          }
        return false;
      }

    }


    /**
     * Method displays all key-value pairs present in HashMapCustom.,
     * insertion order is not guaranteed, for maintaining insertion order
     * refer linkedHashMapCustom.
     * @param key
     */
    public void display(){

       Entry<K, V> currentEntry=header;
       while(currentEntry!=null){
           System.out.print("{"+currentEntry.key+"="+currentEntry.value+"}" +" ");
           currentEntry=currentEntry.after;
       }

    }
    /**
     * Method implements hashing functionality, which helps in finding the appropriate
     * bucket location to store our data.
     * This is very important method, as performance of HashMapCustom is very much
     * dependent on this method's implementation.
     * @param key
     */
    private int hash(K key){
        return Math.abs(key.hashCode()) % capacity;
    }

}


/** Copyright (c), AnkitMittal  JavaMadeSoEasy.com */
/**
 * Main class- to test HashMap functionality.
 */
public class LinkedHashMapCustomApp {

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

```
            LinkedHashMapCustom<Integer, Integer> linkedHashMapCustom = new
LinkedHashMapCustom<Integer, Integer>();

            linkedHashMapCustom.put(21, 12);
            linkedHashMapCustom.put(25, 121);
            linkedHashMapCustom.put(30, 151);
            linkedHashMapCustom.put(33, 15);
            linkedHashMapCustom.put(35, 89);

            System.out.println("Display values corresponding to keys>");
            System.out.println("value corresponding to key 21="
                        + linkedHashMapCustom.get(21));
            System.out.println("value corresponding to key 51="
                        + linkedHashMapCustom.get(51));

            System.out.print("Displaying : ");
            linkedHashMapCustom.display();

            System.out.println("\n\nvalue corresponding to key 21 removed: "
                        + linkedHashMapCustom.remove(21));
            System.out.println("value corresponding to key 22 removed: "
                        + linkedHashMapCustom.remove(22));

            System.out.print("Displaying : ");
            linkedHashMapCustom.display();

    }
}

/*Output

Display values corresponding to keys>
value corresponding to key 21=12
value corresponding to key 51=null
Displaying : {21=12} {25=121} {30=151} {33=15} {35=89}

value corresponding to key 21 removed: true
value corresponding to key 22 removed: false
Displaying : {25=121} {30=151} {33=15} {35=89}

*/
```

6) Complexity calculation of put and get methods in LinkedHashMap >

Complexity offered by put and get methods of LinkedHashMap is same as that of HashMap. **Additionally, for maintaining insertion order during put method -** doubly linked list's **header**, **after and before** are also updated (whichever is needed to be updated).

*6.1) **put** method - **worst Case** complexity >*

O(n) + for maintaining insertion order during put method - doubly linked list's **header**, **after and before** are also updated (whichever is needed to be updated)**.**

*6.2) **put** method - **best Case** complexity >*

O(1). for maintaining insertion order during put method - doubly linked list's **header**, **after and before** are also updated (whichever is needed to be updated)**.**

*6.3) **get** method - **worst Case** complexity >*

O(n)

*6.4) **get** method - **best Case** complexity >*

O(1)