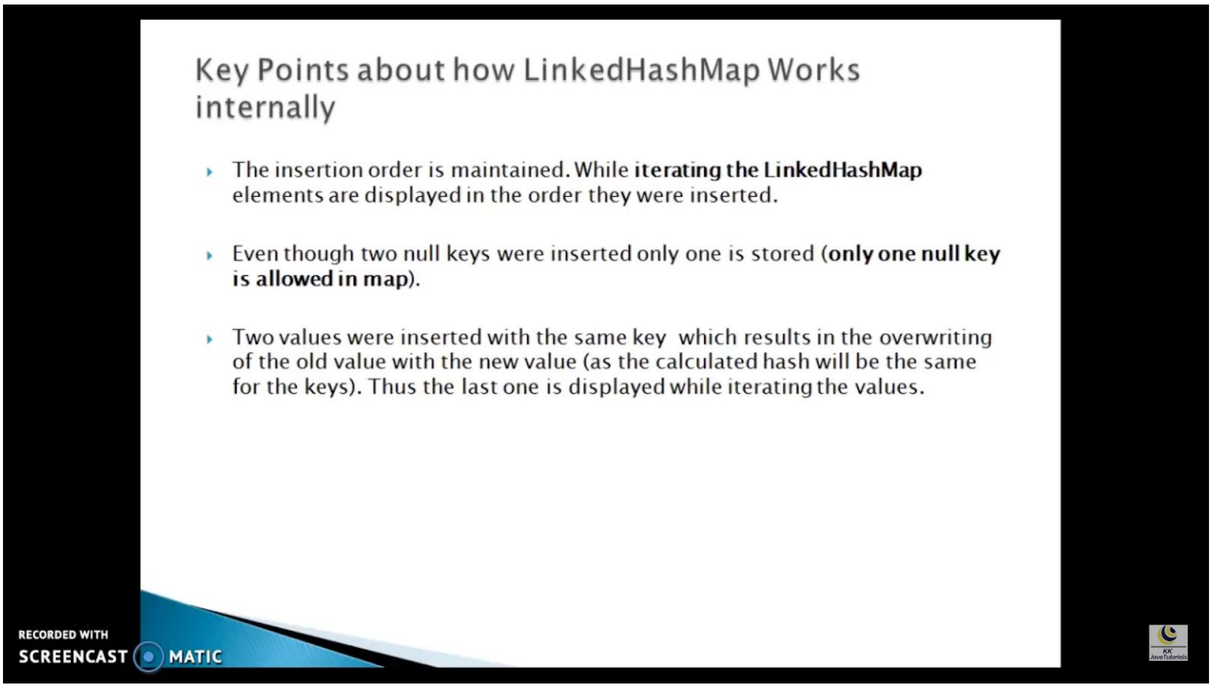
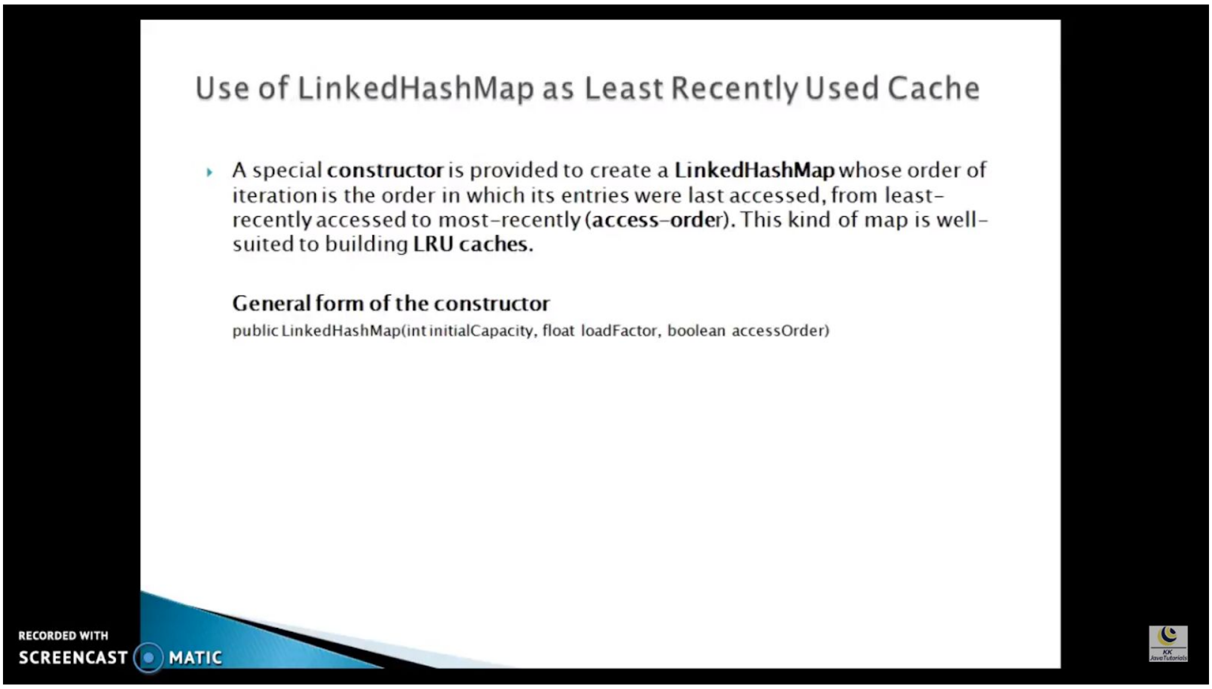
LinkedHashMap



Key Points about how LinkedHashMap Works internally

- The insertion order is maintained. While **iterating the LinkedHashMap** elements are displayed in the order they were inserted.
- Even though two null keys were inserted only one is stored (**only one null key is allowed in map**).
- Two values were inserted with the same key which results in the overwriting of the old value with the new value (as the calculated hash will be the same for the keys). Thus the last one is displayed while iterating the values.

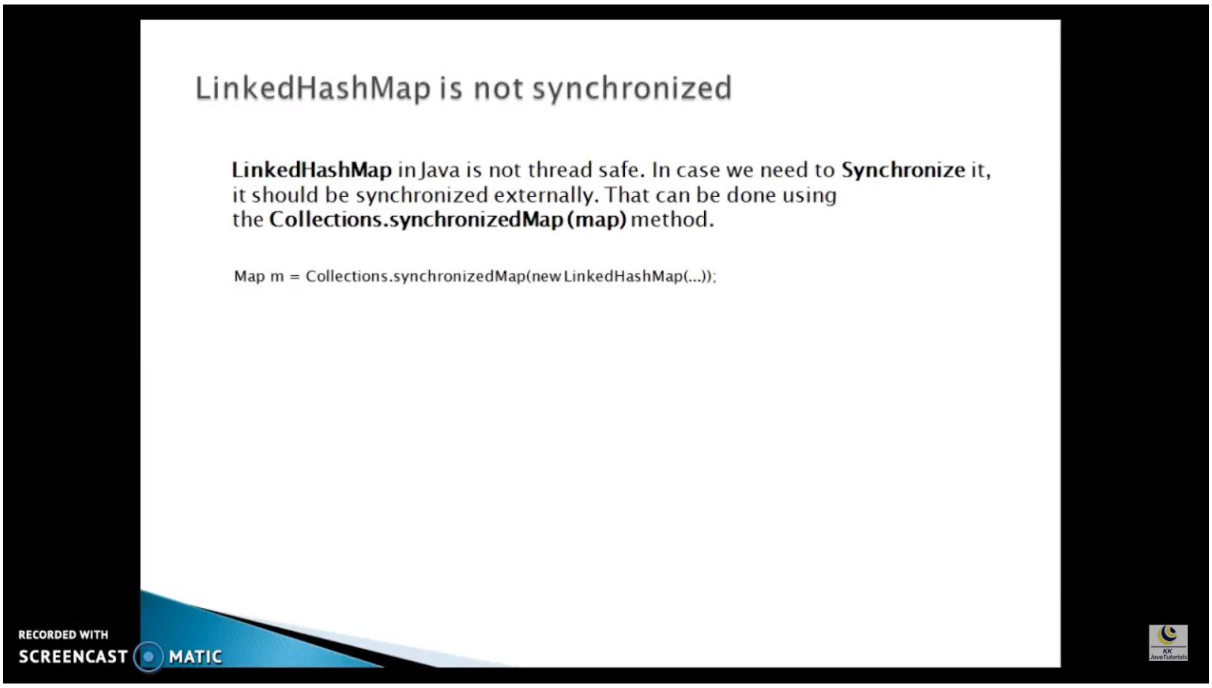


Use of LinkedHashMap as Least Recently Used Cache

- A special **constructor** is provided to create a **LinkedHashMap** whose 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.**

General form of the constructor

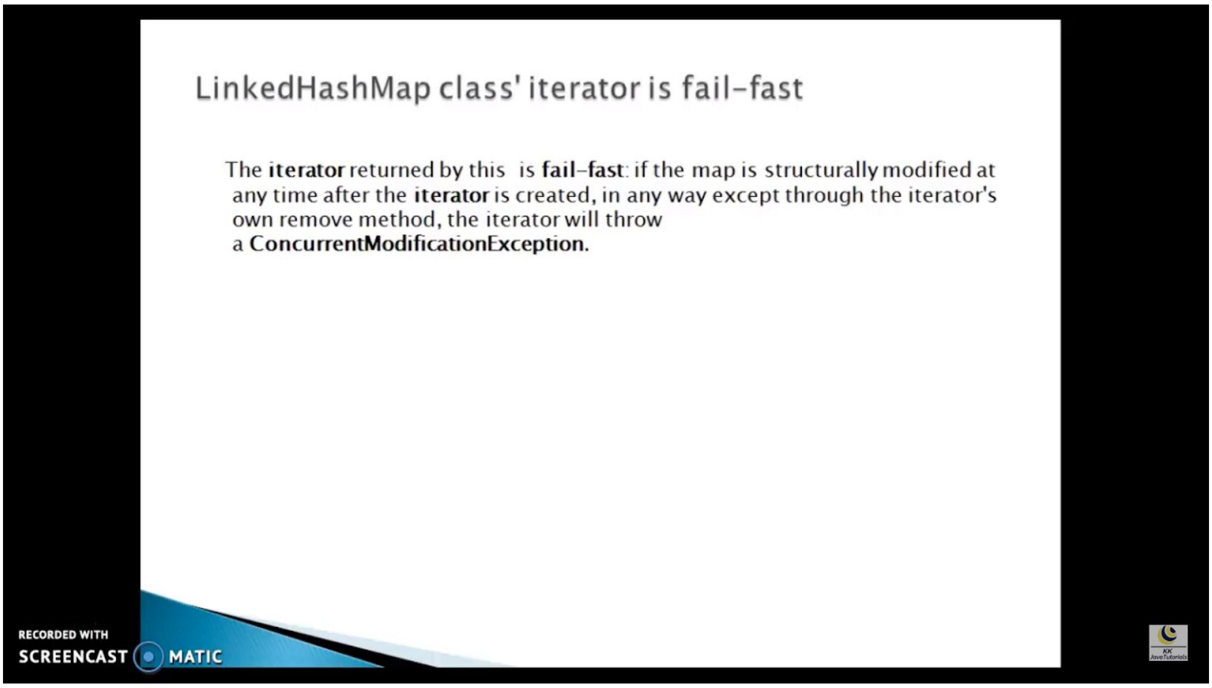
```
public LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)
```



LinkedHashMap is not synchronized

LinkedHashMap in Java is not thread safe. In case we need to **Synchronize** it, it should be synchronized externally. That can be done using the **Collections.synchronizedMap(map)** method.

```
Map m = Collections.synchronizedMap(new LinkedHashMap(...));
```



LinkedHashMap class' iterator is fail-fast

The **iterator** returned by this is **fail-fast**: if the map is structurally modified at any time after the **iterator** is created, in any way except through the iterator's own remove method, the iterator will throw a **ConcurrentModificationException.**

inkedHashMap is just an extension of HashMap as the class definition is as below

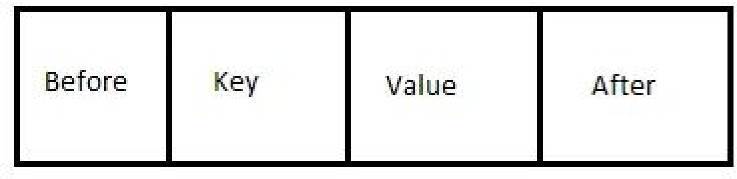
```
public class LinkedHashMap<K, V>
extends HashMap<K, V>
implements Map<K, V>
```

Structure of Each Node

Internally, the node of the *LinkedHashMap* represents as the below:

- int hash
- K key
- V value
- Node next
- Node previous

Let's see the working of *LinkedHashMap* diagrammatically.



Before	Key	Value	After

Hashing Implementation

As we have discussed the Hashing implementation of the *LinkedHashMap* same as the *HashMap* hashing implementation, we have already discussed in **this article**. Let's see the following class of the Key implementations:

```
public class Key {

        final int data = 112;
        private String key;

        public Key(String key) {
                super();
                this.key = key;
        }

        //index = hashCode(key) & (n-1).

        @Override
        public int hashCode() {
                final int prime = 31;
                int result = 1;
                result = prime * result + data;
                result = prime * result + ((key == null) ? 0 : (key.charAt(0)+"").hashCode());
                System.out.println("hashCode for key: "+ key + " = " + result);
```

```
		System.out.println("Index "+ (result & 15));
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Key other = (Key) obj;
		if (data != other.data)
			return false;
		if (key == null) {
			if (other.key != null)
				return false;
		} else if (!key.equals(other.key))
			return false;
		return true;
	}

}
```

In the above code, I am taking a class Key and override *hashCode()* method to show different scenarios. Here, overridden *hashCode()* method return a calculated hashcode. The *hashCode()* method is used to get the hash code of an object. The *hashCode()* method of object class returns the memory reference of the object in integer form. But In *LinkedHashMap*, *hashCode()* is used to calculate the bucket and therefore calculate the index.

We have also overridden *equals()* method in the above code, equals method is used to check that 2 objects are equal or not. *LinkedHashMap* uses *equals()* to compare the key whether they are equal or not.

Bucketing and Indexing

Finding bucket in the *LinkedHashMap* is same as the *HashMap*, it uses hash code to find bucket as the following:

Let’s see the following relationship between bucket and capacity is as follows

```
capacity = number of buckets * load factor
```

If the hash code of two items is same then both items will be stored into the same bucket.

```
index = hashCode(key) & (n-1).
```

In the above formula, n is a number of buckets or the size of an array. In our example, I will consider n as default size that is 16.

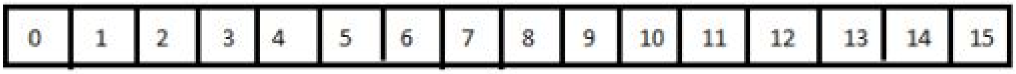
Let’s see the how does *LinkedHashMap* work internally.

Internal Working of LinkedHashMap in Java

Step 1: Create an empty *LinkedHashMap* as the following

```
Map map = new LinkedHashMap();
```

The default size of *LinkedHashMap* is taken as 16 as the following empty array with size 16.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

You can see the above image initially there is no element in the array.

Step 2: Inserting first element Key-Value Pair as the below:

```
map.put(new Key("Dinesh"), "Dinesh");
```

This step will be executed as the following:

1. First, it will calculate the hash code of Key {“Dinesh”}. As we have implemented *hashCode()* method for the Key class, hash code will be generated as 4501.
2. Calculate index by using a generated hash code, according to the index calculation formula, it will be 5.

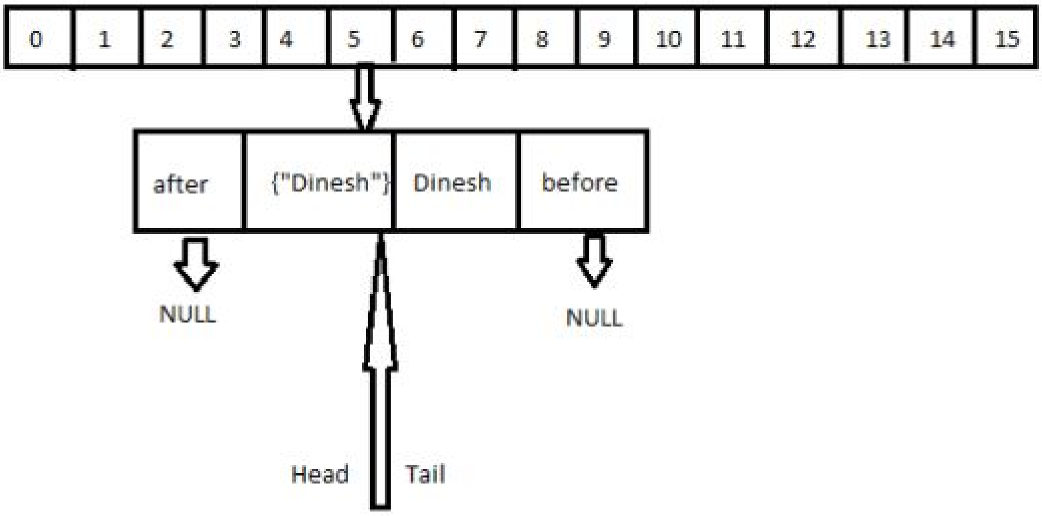
Now it creates a node object as the following:

```
{
  int hash = 4501
```

```
 Key key = {"Dinesh"}
 Integer value = "Dinesh"
 Node before = null
 Node after = null
}
```

3.
4. This node will be placed at index 5. As of now, we are supposing there is no node present at this index because it is a very first element.
5. As *LinkedHashMap* will provide us with the retrieval order as insertion order of elements, it needs to keep track of last inserted object. It has two references head and tail which will keep track of the latest object inserted and the first object inserted. Because it is very first node i.e. {“Dinesh”} is added then head and tail will both point to the same object.

Let’s see the following diagram of the *LinkedHashMap*:



October 8, 2018

4 COMMENTS

Internal Working of LinkedHashMap in Java

Previous

Next

In the previous article, we have discussed **internal working about the *HashMap*** and here we will discuss the internal working of *LinkedHashMap* in **Java**. As we know that, a lot of interviewers ask internal working of data structures such ***HashMap***, ***TreeMap***, ***LinkedHashMap***, ***LinkedList*** etc. That is why I have brought such questions in front of you.

LinkedHashMap extends *HashMap* class and implements *Map* interface. That means *LinkedHashMap* has all functionality same as the *HashMap* either calculating index using Hashing for bucket finding. The difference between *LinkedHashMap* and *HashMap* is the *LinkedHashMap* has retrieval order same as insertion order.

LinkedHashMap in Java

LinkedHashMap is just an extension of HashMap as the class definition is as below

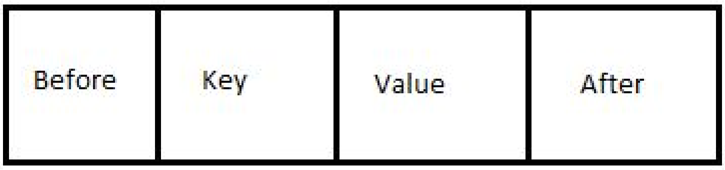
```
public class LinkedHashMap<K, V>
extends HashMap<K, V>
implements Map<K, V>
```

Structure of Each Node

Internally, the node of the *LinkedHashMap* represents as the below:

- int hash
- K key
- V value
- Node next
- Node previous

Let’s see the working of *LinkedHashMap* diagrammatically.



LinkedList Algorithms Interview Questions

Find and Break a Loop in a Linked list

How to Detect loop in a linked list

Find the nth node from the end of a singly linked list

Find the middle element in a linked list

How to Reverse linked list in Java

Delete given node from a singly linked list

Remove Duplicates from the Unsorted Singly Linked list

The singly linked list is palindrome without extra space

Hashing Implementation

As we have discussed the Hashing implementation of the *LinkedHashMap* same as the *HashMap* hashing implementation, we have already discussed in **this article**. Let’s see the following class of the Key implementations:

```
package com.dineshonjava.algo.map;

/**
 * @author Dinesh.Rajput
 *
 */
public class Key {

        final int data = 112;
        private String key;

        public Key(String key) {
                super();
                this.key = key;
        }

        //index = hashCode(key) & (n-1).
```

```
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + data;
        result = prime * result + ((key == null) ? 0 : (key.charAt(0)+"").hashCode());
        System.out.println("hashCode for key: "+ key + " = " + result);
        System.out.println("Index "+ (result & 15));
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Key other = (Key) obj;
        if (data != other.data)
            return false;
        if (key == null) {
            if (other.key != null)
                return false;
        } else if (!key.equals(other.key))
            return false;
        return true;
    }
}
```

In the above code, I am taking a class Key and override *hashCode()* method to show different scenarios. Here, overridden *hashCode()* method return a calculated hashcode. The *hashCode()* method is used to get the hash code of an object. The *hashCode()* method of object class returns the memory reference of the object in integer form. But In *LinkedHashMap*, *hashCode()* is used to calculate the bucket and therefore calculate the index.

We have also overridden *equals()* method in the above code, equals method is used to check that 2 objects are equal or not. *LinkedHashMap* uses *equals()* to compare the key whether they are equal or not.

Bucketing and Indexing

Finding bucket in the *LinkedHashMap* is same as the *HashMap*, it uses hash code to find bucket as the following:

Let’s see the following relationship between bucket and capacity is as follows

capacity = number of buckets * load factor

If the hash code of two items is same then both items will be stored into the same bucket.

index = hashCode(key) & (n-1).

In the above formula, n is a number of buckets or the size of an array. In our example, I will consider n as default size that is 16.

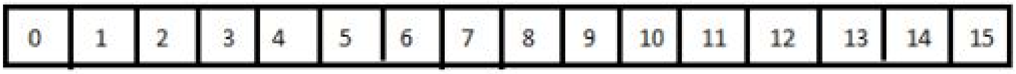
Let’s see the how does *LinkedHashMap* work internally.

Internal Working of LinkedHashMap in Java

Step 1: Create an empty *LinkedHashMap* as the following

```
Map map = new LinkedHashMap();
```

The default size of *LinkedHashMap* is taken as 16 as the following empty array with size 16.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

You can see the above image initially there is no element in the array.

Step 2: Inserting first element Key-Value Pair as the below:

```
map.put(new Key("Dinesh"), "Dinesh");
```

This step will be executed as the following:

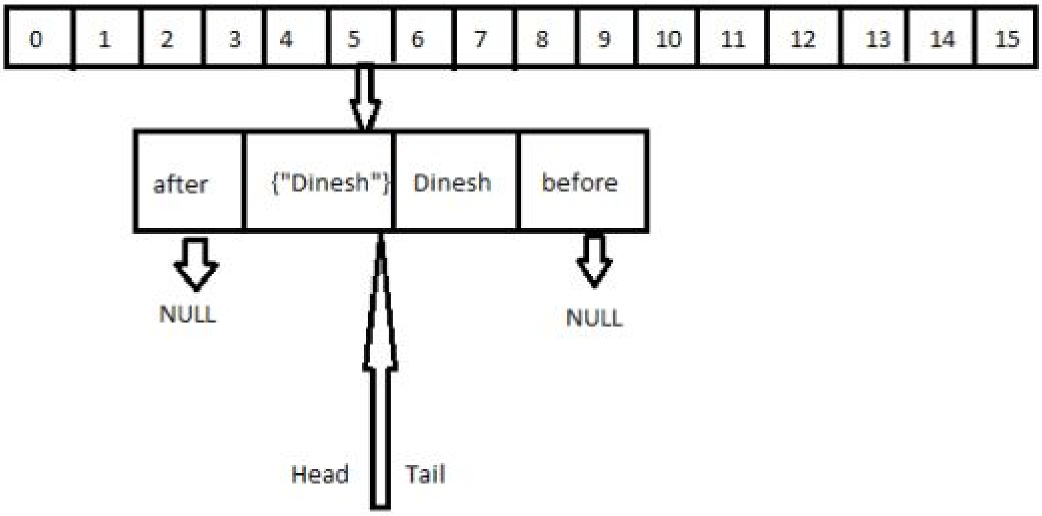
1. First, it will calculate the hash code of Key {“Dinesh”}. As we have implemented *hashCode()* method for the Key class, hash code will be generated as 4501.
2. Calculate index by using a generated hash code, according to the index calculation formula, it will be 5.

Now it creates a node object as the following:

```
{
 int hash = 4501
 Key key = {"Dinesh"}
 Integer value = "Dinesh"
 Node before = null
 Node after = null
}
```

3.
4. This node will be placed at index 5. As of now, we are supposing there is no node present at this index because it is a very first element.
5. As *LinkedHashMap* will provide us with the retrieval order as insertion order of elements, it needs to keep track of last inserted object. It has two references head and tail which will keep track of the latest object inserted and the first object inserted. Because it is very first node i.e. {“Dinesh”} is added then head and tail will both point to the same object.

Let’s see the following diagram of the *LinkedHashMap*:



Step 3: Adding another element Key-Value Pair as the below:

```
map.put(new Key("Anamika"), "Anamika");
```

This step will be executed as the following:

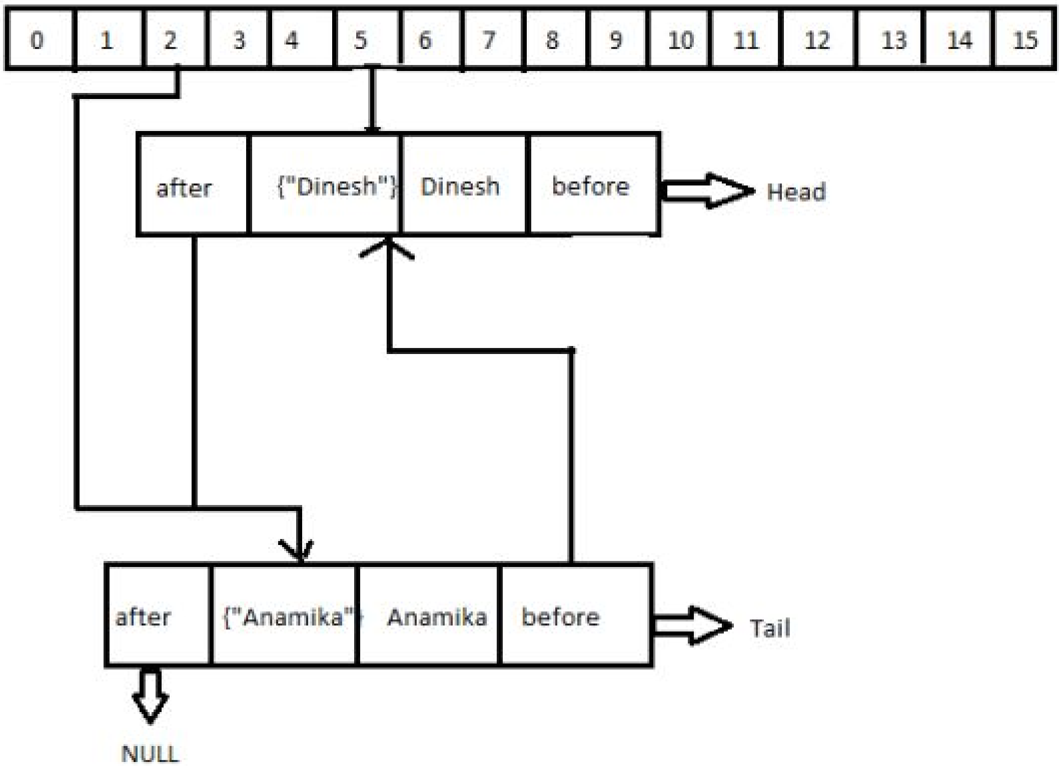
1. First, it will calculate the hash code of Key {“Anamika”}. As we have implemented hashCode() method for the Key class, hash code will be generated as 4498.
2. Calculate index by using a generated hash code, according to the index calculation formula, it will be 2.

But, as we are inserting {“Anamika”} after {“Dinesh”}, this information must be present in form of some links.
So {“Dinesh”}.after will refer to {“Anamika”}, {“Anamika”}.before will refer to {“Dinesh”}. The head refers to {“Dinesh”} and tail refers to {“Anamika”}. Now it creates a node object as the following:

```
{
 int hash = 4498
 Key key = {"Anamika"}
 Integer value = "Anamika"
 Node after = null
 Node before = {“Dinesh”}
}
```

3.
4. This node will be placed at index 2. As of now, we are supposing there is no node present at this index because it is a very first element.

Let’s see the following diagram of the *LinkedHashMap*:



Step 4: Adding another element Key-Value Pair as the below:

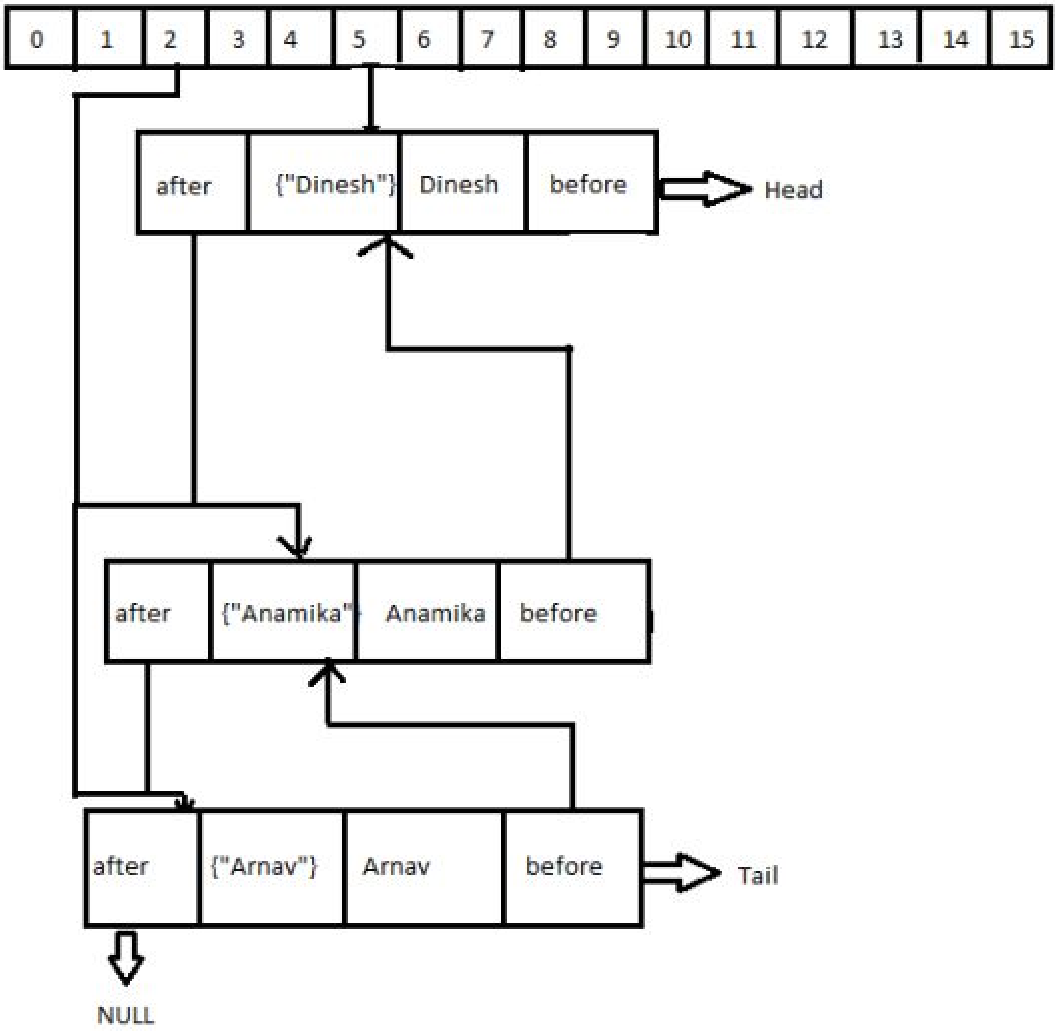
map.put(new Key("Arnav"), "Arnav");

This step will be executed as the following:

1. First, it will calculate the hash code of Key {“Arnav”}. As we have implemented *hashCode()* method for the Key class, hash code will be generated as 4498.

2. Calculate index by using a generated hash code, according to the index calculation formula, it will be 2.
3. {“Anamika”} will have next object as {“Arnav”}
4. {“Arnav”} will have the previous object as {“Anamika”}
5. The head will remain unchanged and will refer to {“Dinesh”}
6. The tail will refer to latest added {“Arnav”}

Let’s see the following diagram of the LinkedHashMap:



I hope the article helped to understand the working of *put()* method of LinkedHashMap in java.

Let’s see the *LinkedHashMap’s get()* method internal working as below:

LinkedHashMap’s get() method work internally

Now we will fetch a value from the *LinkedHashMap* using the *get()* method. As the following, we are fetching the data for key {“Arnav”}:

```
map.get(new Key("Arnav"));
```

As we know that *LinkedHashMap* and *HashMap* have almost the same functionality except maintaining insertion order. So fetching data from *LinkedHashMap* has the same steps as HashMap.