eazy bytes

Map

In Java, the Map interface is used to represent a mapping between keys and values. It is an abstract data type that provides a way to store, retrieve, and manipulate data based on keys. The Map interface is implemented by several classes but HashMap, TreeMap, LinkedHashMap are the most commonly used map implementations.

Suppose you want to store information about countries and their respective capitals; in that case, utilizing a Map would be an appropriate choice.

KEY	VALUE
India	New Delhi
USA	Washington, DC

Few important points about Map:

- Map is part of the Java Collections Framework, but it does not extend the Collection interface
- Map is a parameterized type with two type variables, Map<K, V>. Type variable K represents the type of keys held by the map, and type variable V represents the type of the values that the keys are mapped to. For example, a mapping from String keys to Person values, can be represented with a Map<String,Person>.
- Map.Entry is a nested interface defined within Map: it simply represents a single key/value pair.
- The most important Map methods are put(), which adds a key/value pair in the map; get(), which queries the value associated with a specified key; and remove(), which removes the specified key and its associated value from the map.

HashMap

eazy bytes

Creating HashMap

```
// Creating a new HashMap
Map<String, String> countryMap = new HashMap<>();

// Add some key-value pairs to the map
countryMap.put("India", "New Delhi");
countryMap.put("USA", "Washington, DC");
countryMap.put("France", "Paris");

// Fetch a value based on the Key
String capital = countryMap.get("India");
```

Here are some important features of HashMap:

- HashMap stores key-value pairs in a hash table, which uses a hash function to compute an index into an array of buckets. That's why we can also say HashMap works on the principle of hashing
- The hash function is used to map the key to an index in the hash table. The index is then used to store the value associated with the key.
- HashMap allows null values and null keys. We cannot store duplicate keys in HashMap. However, if you try to store duplicate key with another value, it will replace the value.
- HashMap is not synchronized, meaning that it is not thread-safe. The Collections.synchronizedMap(map) is used to synchronize HashMap and make them thread safe as well.
- HashMap is not ordered, meaning that it does not maintain the order of the elements in the map.

How HashMap Store Key, Value

eazy bytes

Let's take the below code,

```
countryMap.put("India", "New Delhi");
```

To identify the bucket index number where it has to store the elements provided, HashMap follows the below steps,

1) Find hashCode by invoking the hashCode() method of the key,

```
"India".hashCode() -> Assume hashCode as 3492
```

2) Find Bucket index by considering hashCode and length of the hashmap. Let's say the index is calculated as 3

// By following similar steps, for France, the index is going to be calculated as 7

```
countryMap.put("France", "Paris");
```

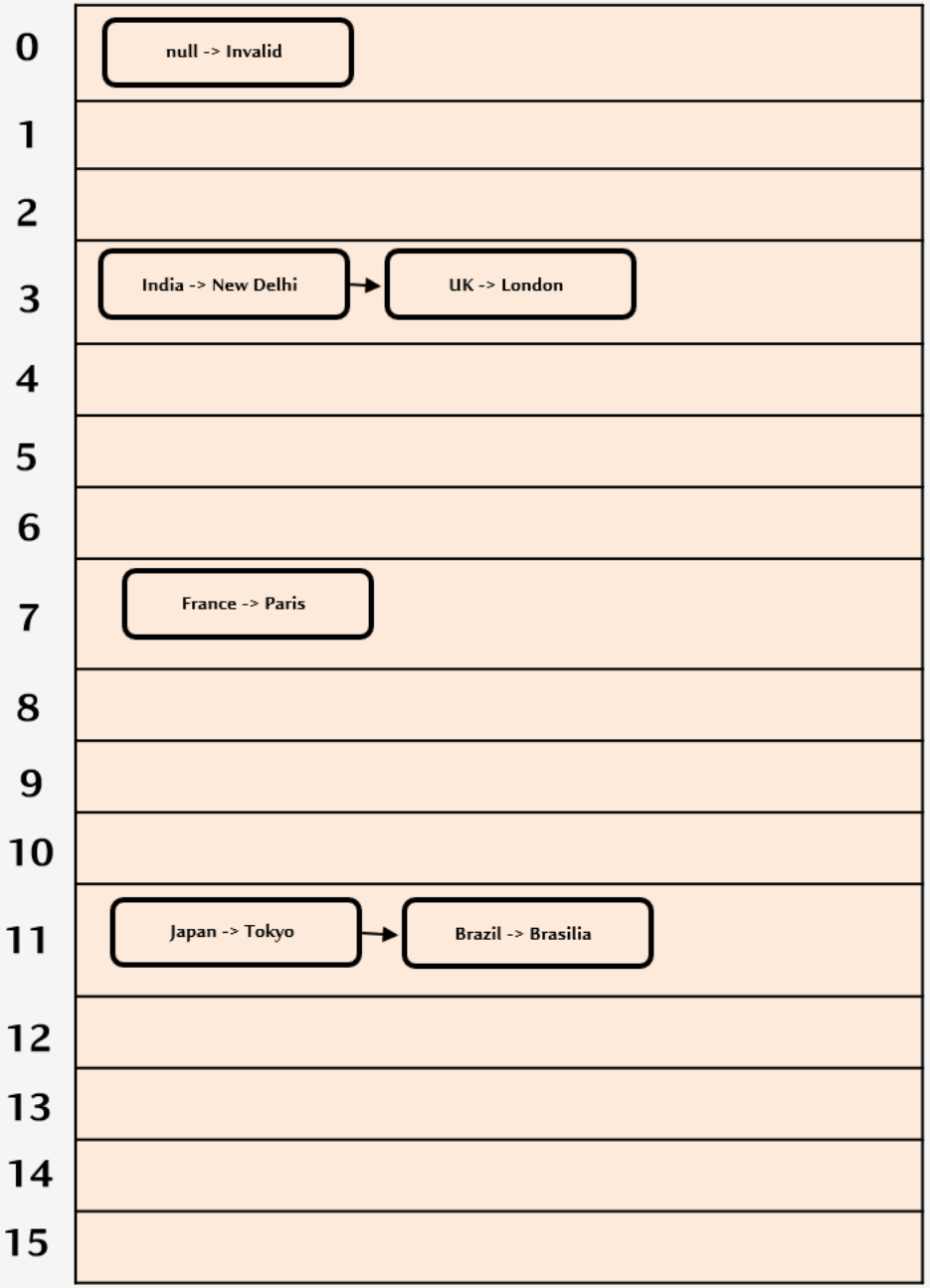
// By following similar steps, for Japan, the index is going to be calculated as 11

```
countryMap.put("Japan", "Tokyo");
```

We can store up to a single null key and it is always going to stored at the index 0

```
countryMap.put(null, "Invalid");
```

How HashMap Store Key, Value



eazy bytes

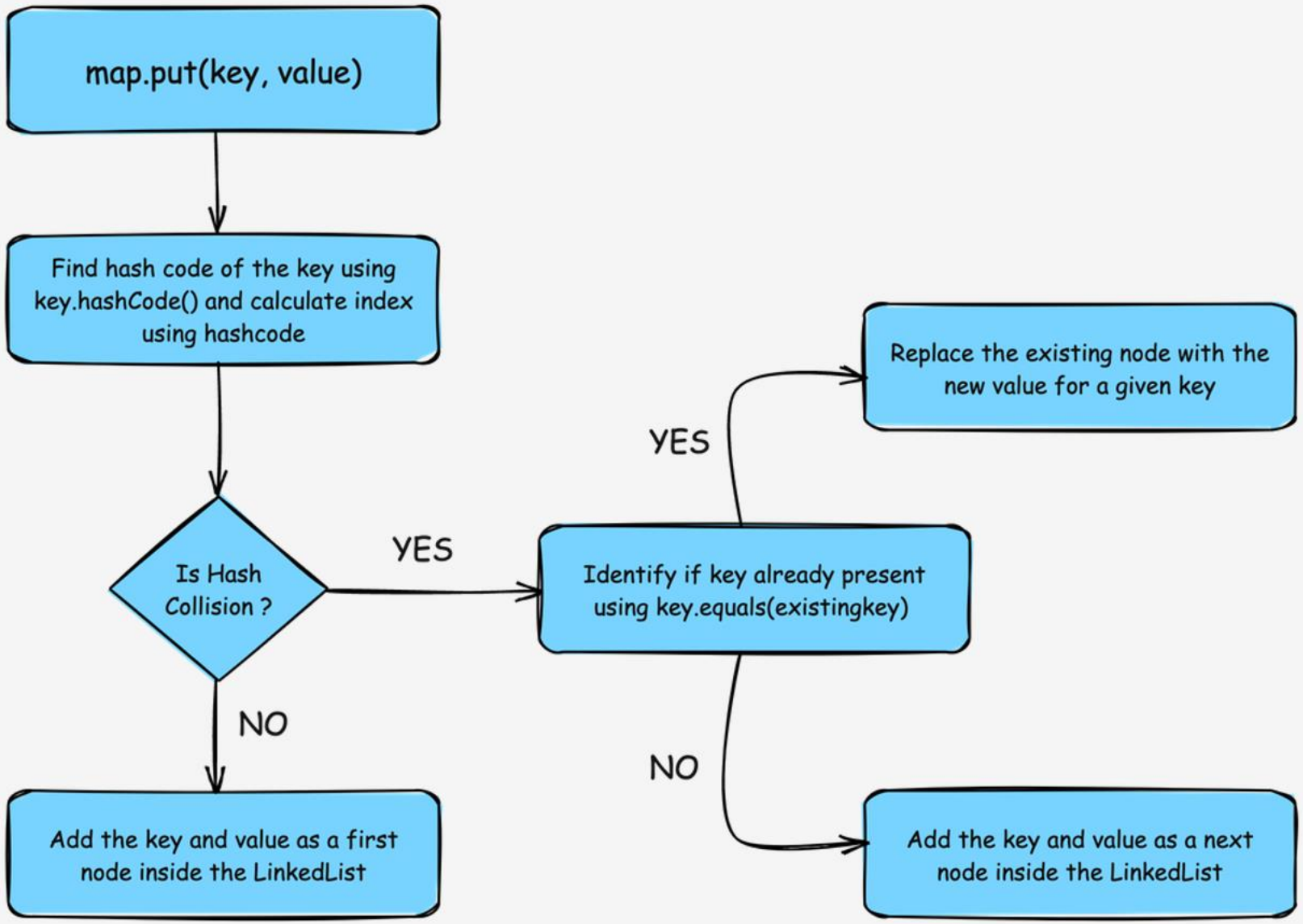
In a HashMap, **hash collisions** occur when two or more keys produce the same hash code. Hash codes are used to determine the index or bucket in which a key-value pair should be stored. Ideally, each key should have a unique hash code, but due to the finite range of hash codes and the potentially infinite number of keys, collisions can happen.

When a collision occurs, HashMap employs a strategy to handle it. One common strategy is to use a data structure called a linked list or another data structure like a binary tree to store multiple key-value pairs in the same bucket. Each element in the bucket is associated with a different key but produces the same hash code.

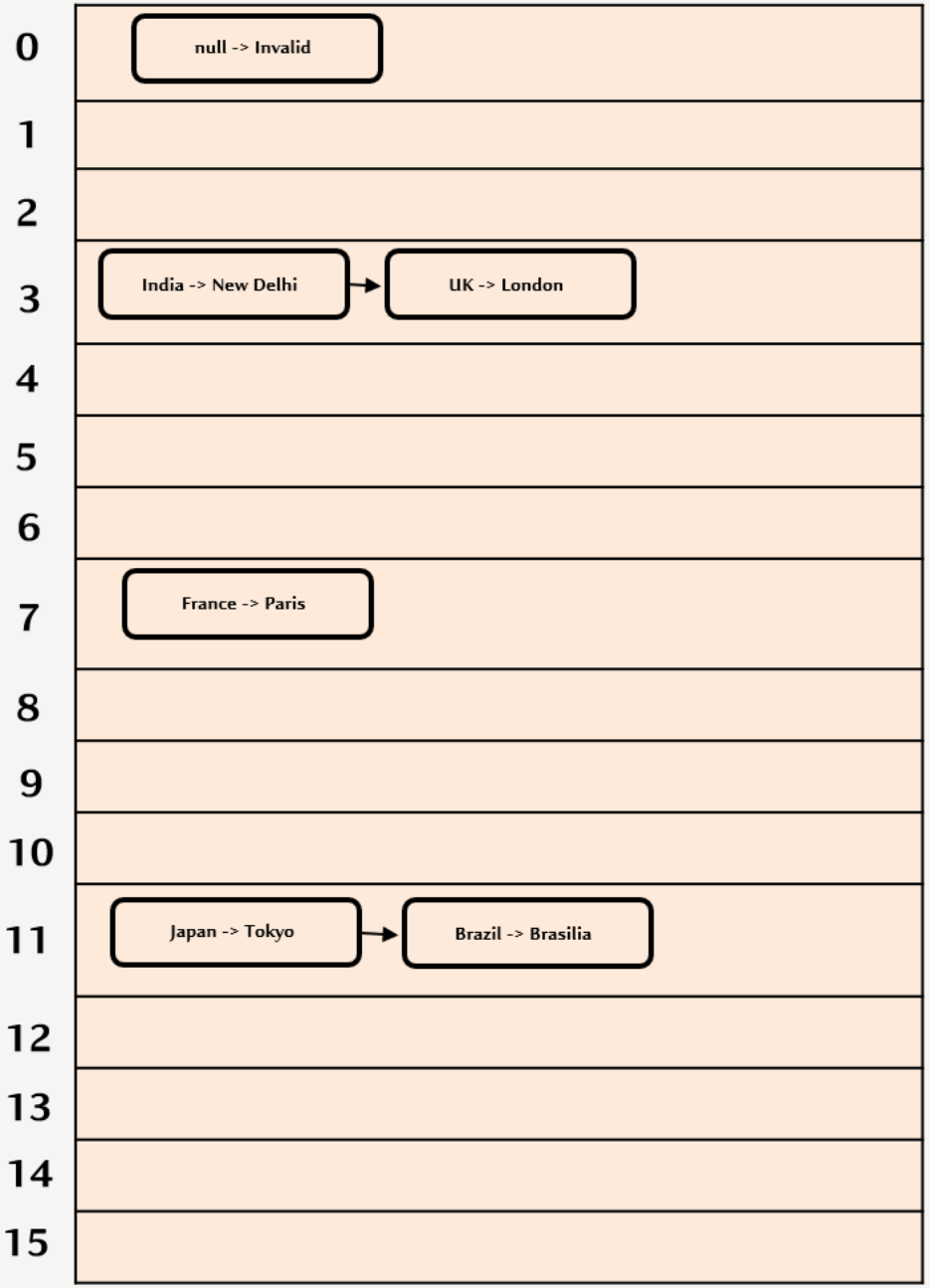
```
// Assume that the hashCode for UK is 3492 which is same as for India. Since the hashCodes are same, for UK as well the bucket index is going to be calculated as 3
countryMap.put(”UK", ”London");

// Assume that the hashCode for Brazil is same as for Japan. Since the hashCodes are same, for Brazil as well the bucket index is going to be calculated as 11
countryMap.put(”Brazil", ”Brasilia");
```

How HashMap Store Key, Value

eazy bytes



How HashMap retrieve Value



eazy bytes

To obtain an object from a HashMap, you use the get() method and provide the key object as an argument. This causes the key object to generate the same hash code as before (which is necessary for retrieving the object), and directs us to the same bucket location. If there is only one object in that bucket, it will be returned as the value object that was previously stored. It is important for HashMap keys to be immutable, such as Strings, in order for this process to work properly.

// Below code will return the value as Paris with out any collision

```
countryMap.get("France");
```

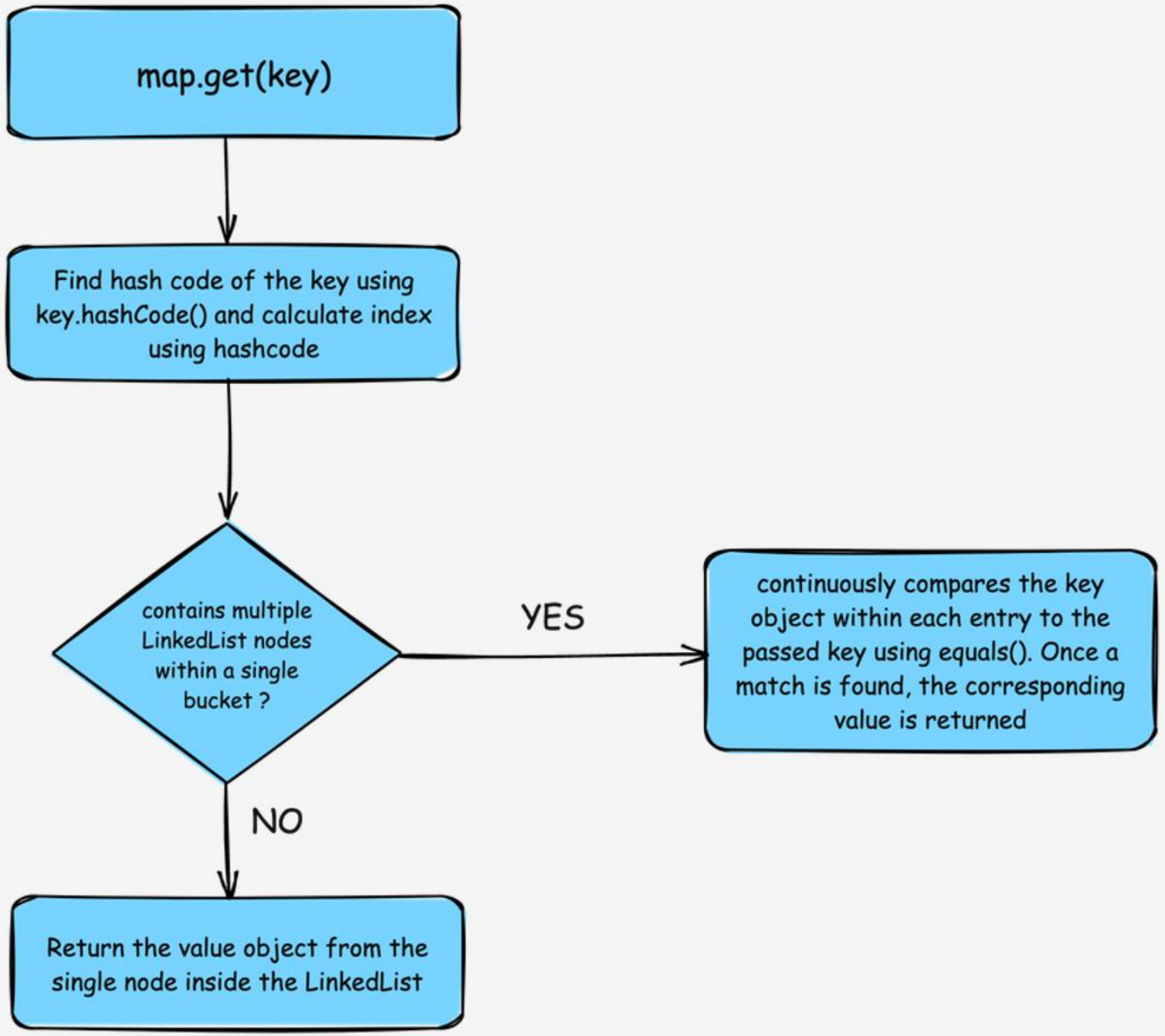
When attempting to retrieve an object from a HashMap that contains multiple LinkedList nodes within a single bucket or index, the HashMap will conduct an additional check by searching for the correct value. This is achieved through the use of the equals() method. Each node in the LinkedList contains an entry, and HashMap continuously compares the key object within each entry to the passed key using equals(). Once a match is found, the corresponding value is returned by the Map.

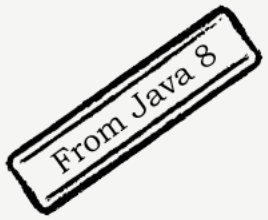
// Below code will result in collision & return the value as London

```
countryMap.get("UK");
```

How HashMap retrieve Value

eazy bytes





HashMap improvements in Java 8

eazy bytes

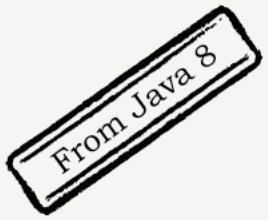
> In Java 8, HashMap was improved to include a new data structure called TreeNodes. This was done to improve performance in cases where the HashMap contains a large number of entries with the same hash code, causing collisions and the creation of long linked lists within the HashMap buckets.

TreeNodes work by creating a self-balancing binary tree within the bucket. When new entries are added to the bucket, the tree is restructured as needed to maintain balance and ensure that retrieval times remain fast. In addition to improving performance, the use of TreeNodes also reduces the likelihood of collisions, as entries with the same hash code are distributed more evenly throughout the tree.

Because TreeNodes are about twice the size of regular nodes, they get used only when bins contain enough nodes to warrant use. And when they become too small (due to removal or resizing) they are converted back to plain bins. In usages with well-distributed user hashCodes, tree bins are rarely used.

Think like you are trying to store the below key and values and for a second imagine they all have same hashcode which will make them to store at a same index or bucket,

```
4, “Four”
1, “One”
7, “Seven”
2, “Two”
15, “Fifteen”
11, “Eleven”
9, “Nine”
13, “Thirteen”
5, “Five”
```

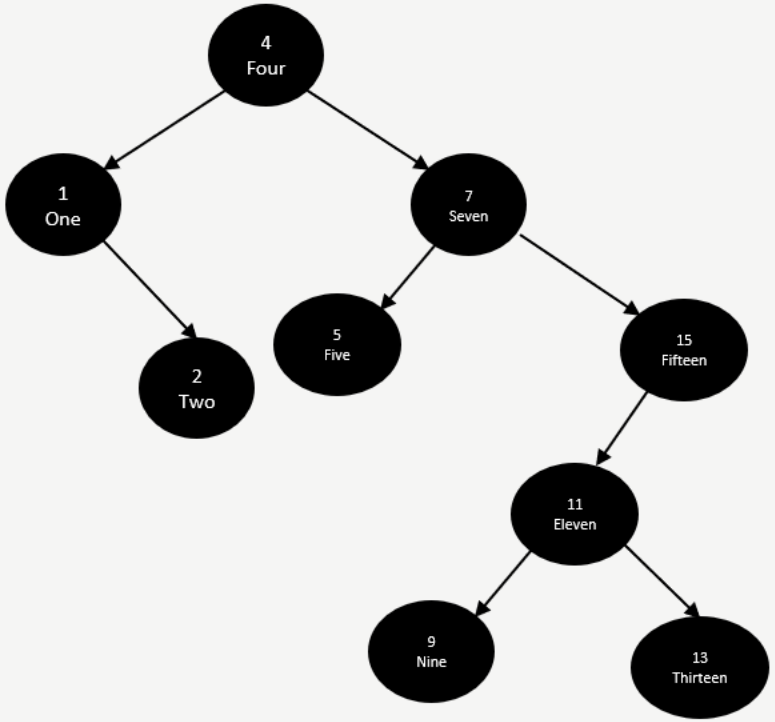


HashMap improvements in Java 8

eazy bytes

Below is how the Map entries are stored inside a normal LinkedList format under a HashMap single bucket. Using this approach, accessing an element is going to take time as the number of entries in a single bucket grows,

Now lets try to see how the same information is stored using Tree Nodes,



Since the tree nodes stores entries using a sorting logic, accessing them will be much quicker compared to LinkedList

```
static final int TREEIFY_THRESHOLD = 8;
static final int UNTREEIFY_THRESHOLD = 6;
```

Based on the above constant fields, the process of converting from List to Tree and viceversa is going to happen.

Iterating HashMap

eazy bytes

There are several ways to iterate through a HashMap in Java. Some of the commonly used methods are:

1 Using keySet() method: The keySet() method returns a set of all the keys present in the HashMap. We can use the iterator() method to get an iterator for the key set and use it to iterate through the HashMap. We can use for-each as well

```
HashMap<String, String> countryMap = new HashMap<>();
countryMap.put("India", "New Delhi");
countryMap.put("USA", "Washington, DC");
countryMap.put("France", "Paris");

Set<String> keys = countryMap.keySet();
Iterator<String> iterator = keys.iterator();
while (iterator.hasNext()) {
    String key = iterator.next();
    String capital = countryMap.get(key);
    // perform required operation on the key-value pair
}
```

2 Using entrySet() method: The entrySet() method returns a set of all the key-value pairs in the HashMap. We can use the iterator() method to get an iterator for the entry set and use it to iterate through the HashMap. We can use for-each as well

```
HashMap<String, String> countryMap = new HashMap<>();
countryMap.put("India", "New Delhi");
countryMap.put("USA", "Washington, DC");
countryMap.put("France", "Paris");

Set<Map.Entry<String, String>> entries = countryMap.entrySet();
Iterator<Map.Entry<String, String>> iterator = entries.iterator();
while (iterator.hasNext()) {
    Map.Entry<String, String> entry = iterator.next();
    String key = entry.getKey();
    String value = entry.getValue();
    // perform required operation on the key-value pair
}
```

Iterating HashMap

eazy bytes

There are several ways to iterate through a HashMap in Java. Some of the commonly used methods are:

3

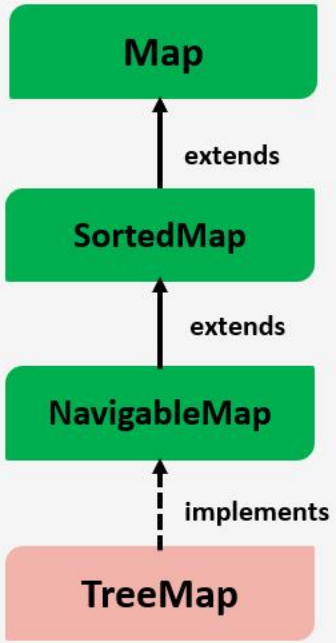
We can iterate only values of the map by invoking values(). We can either use Iterator or for-each to iterate the values of the HashMap

```
// Create a new HashMap
HashMap<String, String> countryMap = new HashMap<>();
countryMap.put("India", "New Delhi");
countryMap.put("USA", "Washington, DC");
countryMap.put("France", "Paris");

for(String value : countryMap.values()){
    System.out.println(value);
}
```

TreeMap

eazy bytes

Java TreeMap class is a red-black tree based implementation & stores key-value pairs in a sorted order based on the keys. This means that when we add key-value pairs to a TreeMap, they are automatically sorted by their keys.




```java
Map<Integer, String> numbers = new TreeMap<>();
numbers.put(23, "Twenty three" );
numbers.put(12, "Twelve" );
numbers.put(42, "Forty two" );
numbers.put(3, "Three" );
numbers.put(19, "Nineteen" );
numbers.put(48, "Forty eight" );
numbers.put(76, "Seventy six" );

for(Map.Entry m:numbers.entrySet())
{
    System.out.println(m.getKey()+"-"+m.getValue());
}
```

output

```
3-Three
12-Twelve
19-Nineteen
23-Twenty three
42-Forty two
48-Forty eight
76-Seventy six
```

TreeMap cannot contain any null key & maintains the ascending order based on key. HashMap does not maintain any order. So, if you need to access the keys in a sorted order, TreeMap is a better choice.

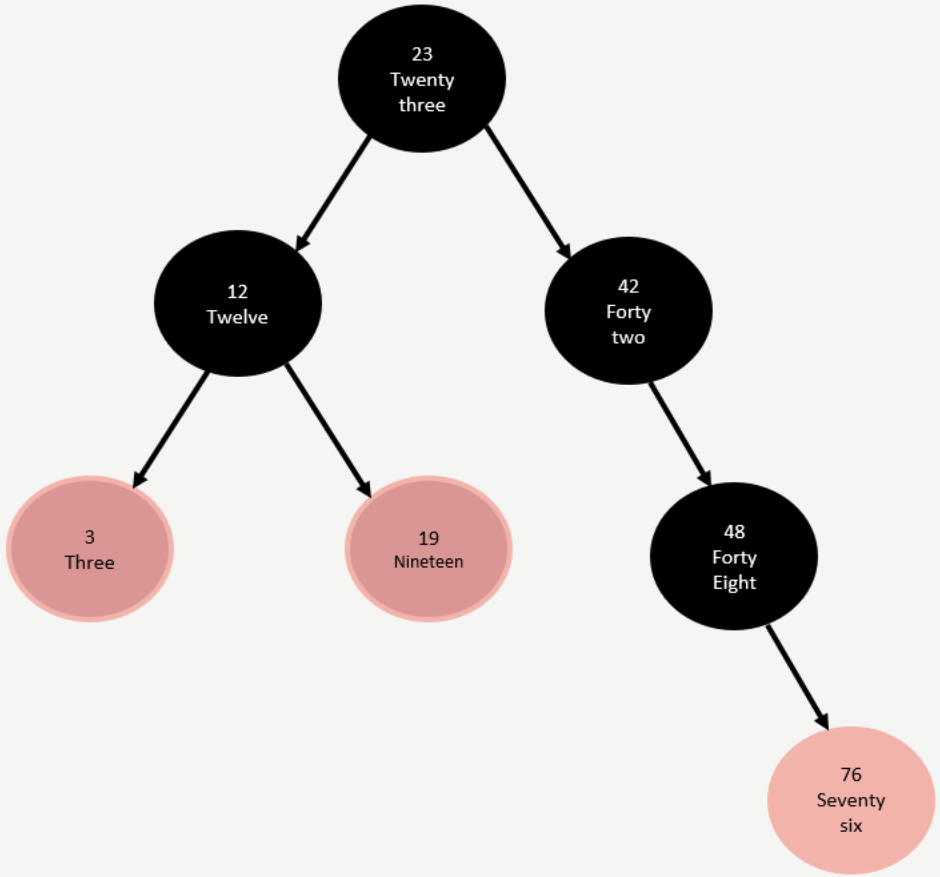
When you need to search for keys or range of keys: TreeMap provides efficient methods for searching keys, finding the first and last keys in the map, and finding keys within a specified range. HashMap does not provide such methods.

How TreeMap store elements

eazy bytes

Here is how the numbers 23, 12, 42, 3, 19, 48, 76 are stored using a Red-Black tree step by step:

```
numbers.put(23, "Twenty three" );

numbers.put(12, "Twelve" );

numbers.put(42, "Forty two" );

numbers.put(3, "Three" );

numbers.put(19, "Nineteen" );

numbers.put(48, "Forty eight" );

numbers.put(76, "Seventy six" );
```



LinkedHashMap

eazy bytes

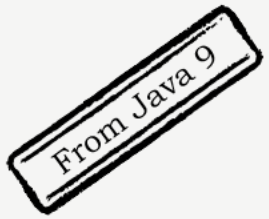
In Java, LinkedHashMap is a class that extends HashMap, implements SequencedMap and maintains the order of elements based on the order of their insertion. It means that when you iterate over the entries of a LinkedHashMap, they are returned in the order in which they were added. LinkedHashMap combines the features of a hash table and a linked list.

```
public class LinkedHashMap<K,V> extends HashMap<K,V>
    implements SequencedMap<K,V>{ }
```

```java
public class LinkedHashMapDemo {
    public static void main(String[] args) {
        // Creating a LinkedHashMap
        Map<Integer, String> linkedHashMap = new LinkedHashMap<>();

        // Adding elements
        linkedHashMap.put(4, "Four");
        linkedHashMap.put(1, "One");
        linkedHashMap.put(7, "Seven");
        linkedHashMap.put(2, "Two");

        // Iterating over the entries
        for (Map.Entry<Integer, String> entry : linkedHashMap.entrySet()) {
            System.out.println(entry.getKey() + ": " + entry.getValue());
        }
    }
}
```

Output:

```
4: Four
1: One
7: Seven
2: Two
```



Creating Immutable Maps

eazy bytes

Immutable Map Static Factory Methods

The Map.of and Map.ofEntries static factory methods provide a convenient way to create immutable maps. A Map cannot contain duplicate keys; each key can map to at most one value. If a duplicate key is detected, then an IllegalArgumentException is thrown. Null values cannot be used as Map keys or values. The syntax of these methods is:

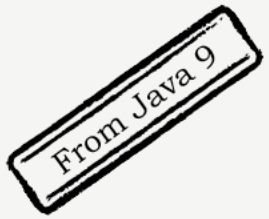
```
Map.of()
Map.of(k1, v1)
Map.of(k1, v1, k2, v2) // fixed-argument form overloads up to 10 key-value pairs
Map.ofEntries(entry(k1, v1), entry(k2, v2),...) // varargs form supports an arbitrary number
                                                // of Entry objects or an array
```

In JDK 8:

```
Map<String, Integer> stringMap = new HashMap<String, Integer>();
stringMap.put("a", 1);
stringMap.put("b", 2);
stringMap.put("c", 3);
stringMap = Collections.unmodifiableMap(stringMap);
```

In JDK 9:

```
Map stringMap = Map.of("a", 1, "b", 2, "c", 3);
```



Creating Immutable Maps

eazy bytes

If you have more than 10 key-value pairs, then create the map entries using the Map.entry method, and pass those objects to the Map.ofEntries method. For example:

```
import static java.util.Map.entry;
Map <Integer, String> friendMap = Map.ofEntries(
   entry(1, "Tom"),
   entry(2, "Dick"),
   entry(3, "Harry"),
   ...
   entry(99, "Mathilde"));
```

The collections returned by the convenience factory methods added in JDK 9 are conventionally immutable. Any attempt to add, set, or remove elements from these collections causes an UnsupportedOperationException to be thrown.