HashMap in Java

HashMap is a part of Java's collection since Java 1.2. It provides the basic implementation of Map interface of Java. It stores the data in (Key, Value) pairs. To access a value one must know its key. HashMap is known as HashMap because it uses a technique called Hashing. Hashing is a technique of converting a large String to small String that represents the same String. A shorter value helps in indexing and faster searches. HashSet also uses HashMap internally. It internally uses a link list to store key-value pairs already explained in HashSet in detail and further articles.
Few important features of HashMap are:

- HashMap is a part of java.util package.
- HashMap extends an abstract class AbstractMap which also provides an incomplete implementation of Map interface.
- It also implements Cloneable and Serializable interface. K and V in the above definition represent Key and Value respectively.
- HashMap doesn't allow duplicate keys but allows duplicate values. That means A single key can't contain more than 1 value but more than 1 key can contain a single value.
- HashMap allows null key also but only once and multiple null values.
- This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time. It is roughly similar to HashTable but is unsynchronized.

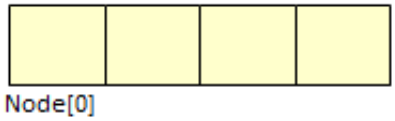
Internal Structure of HashMap

Internally HashMap contains an array of Node and a node is represented as a class which contains 4 fields :

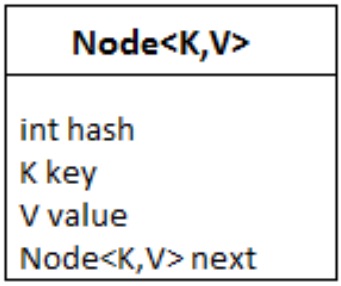
1. int hash
2. K key

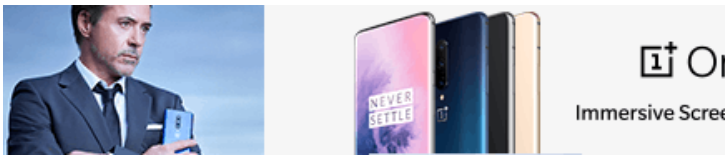
3. V value
4. Node next

It can be seen that node is containing a reference of its own object. So it's a linked list.
HashMap:



Node :





Performance of HashMap

Performance of HashMap depends on 2 parameters:

1. Initial Capacity
2. Load Factor

As already said, Capacity is simply the number of buckets whereas the *Initial Capacity* is the capacity of HashMap instance when it is created. The *Load Factor* is a measure that when rehashing should be done. Rehashing is a process of increasing the capacity. In HashMap capacity is multiplied by 2. Load Factor is also a measure that what fraction of the HashMap is allowed to fill before rehashing. When the number of entries in HashMap increases the product of current capacity and load factor the capacity is increased that is rehashing is done. If the initial capacity is kept higher then rehashing will never be done. But by keeping it higher it increases the time complexity of iteration. So it should be chosen very cleverly to increase performance. The expected number of values should be taken into account to set initial capacity. Most generally preferred load factor value is 0.75 which provides a good deal between time and space costs. Load factor's value varies between 0 and 1.

Synchronized HashMap

As it is told that HashMap is unsynchronized i.e. multiple threads can access it simultaneously. If multiple threads access this class simultaneously and at least one thread manipulates it structurally then it is necessary to make it synchronized externally. It is done by synchronizing some object which encapsulates the map. If No such object exists then it can be wrapped around Collections.synchronizedMap() to make HashMap synchronized and avoid accidental unsynchronized access. As in the following example:

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

Now the Map m is synchronized.

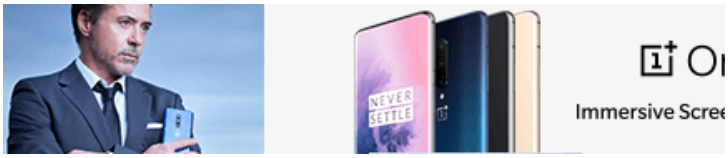
Iterators of this class are fail-fast if any structure modification is done after the creation of iterator, in any way except through the iterator's remove method. In a failure of iterator, it will throw ConcurrentModificationException.

Constructors in HashMap

HashMap provides 4 constructors and access modifier of each is public:

1. **HashMap() :** It is the default constructor which creates an instance of HashMap with initial capacity 16 and load factor 0.75.
2. **HashMap(int initial capacity) :** It creates a HashMap instance with specified initial capacity and load factor 0.75.
3. **HashMap(int initial capacity, float loadFactor) :** It creates a HashMap instance with specified initial capacity and specified load factor.
4. **HashMap(Map map) :** It creates instance of HashMap with same mappings as specified map.

Example:




```
// Java program to illustrate
// Java.util.HashMap
import java.util.HashMap;
import java.util.Map;

public class GFG
{
```

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

        HashMap<String, Integer> map = new HashMap<>();

        print(map);
        map.put("vishal", 10);
        map.put("sachin", 30);
        map.put("vaibhav", 20);

        System.out.println("Size of map is:- " + map.size());

        print(map);
        if (map.containsKey("vishal"))
        {
            Integer a = map.get("vishal");
            System.out.println("value for key \"vishal\" is:- " + a);
        }

        map.clear();
        print(map);
    }

    public static void print(Map<String, Integer> map)
    {
        if (map.isEmpty())
        {
            System.out.println("map is empty");
        }

        else
        {
            System.out.println(map);
        }
    }
}
```

Output:

```
map is empty
Size of map is:- 3
{vaibhav=20, vishal=10, sachin=30}
value for key "vishal" is:- 10
map is empty
```

Time complexity of HashMap

HashMap provides constant time complexity for basic operations, get and put, if hash function is properly written and it disperses the elements properly among the buckets. Iteration over HashMap depends on the capacity of HashMap and number of key-value pairs. Basically, it is directly proportional to the capacity + size. Capacity is the number of buckets in HashMap. So it is not a good idea to keep a high number of buckets in HashMap initially.

Methods in HashMap

1. void clear(): Used to remove all mappings from a map.
2. boolean containsKey(Object key): Used to return True if for a specified key, mapping is present in the map.
3. boolean containsValue(Object value): Used to return true if one or more key is mapped to a specified value.
4. Object clone(): It is used to return a shallow copy of the mentioned hash map.
5. boolean isEmpty(): Used to check whether the map is empty or not. Returns true if the map is empty.
6. Set entrySet(): It is used to return a set view of the hash map.
7. Object get(Object key): It is used to retrieve or fetch the value mapped by a particular key.
8. Set keySet(): It is used to return a set view of the keys.
9. int size(): It is used to return the size of a map.
10. Object put(Object key, Object value): It is used to insert a particular mapping of key-value pair into a map.
11. putAll(Map M): It is used to copy all of the elements from one map into another.
12. Object remove(Object key): It is used to remove the values for any particular key in the Map.
13. Collection values(): It is used to return a Collection view of the values in the HashMap.

Related Articles:

- Hashmap vs Treemap
- Hashmap vs HashTable
- **compute?(K key, BiFunction<K,V> remappingFunction):** This method Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
- **computeIfAbsent?(K key, Function<K> mappingFunction):** This method If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null.
- **computeIfPresent?(K key, BiFunction<K,V> remappingFunction):** This method If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.
- **forEach?(BiConsumer<K,V> action):** This method Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
- **getOrDefault?(Object key, V defaultValue):** This method returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.

- **merge?(K key, V value, BiFunction<K,V> remappingFunction):** This method If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
- **putIfAbsent?(K key, V value):** This method If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.
- **replace?(K key, V value):** This method replaces the entry for the specified key only if it is currently mapped to some value.
- **replace?(K key, V oldValue, V newValue):** This method replaces the entry for the specified key only if currently mapped to the specified value.
- **replaceAll?(BiFunction<K,V> function):** This method replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.

Recent articles on Java HashMap!

This article is contributed by **Vishal Garg**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Recommended Posts:

ArrayList vs. HashMap in Java

HashMap get() Method in Java

HashMap put() Method in Java

HashMap and TreeMap in Java

Traverse through a HashMap in Java

Hashmap vs WeakHashMap in Java

Initialize HashMap in Java

HashMap clear() Method in Java

HashMap containsKey() Method in Java

HashMap clone() Method in Java

HashMap containsValue() Method in Java