Browse Category

Last Updated: Jul 7, 2025
Easy

HashMap in Java

Author
Shreya Deep

Share
2 upvotes

Table of contents

Introduction

HashMap in Java is a data structure that stores key-value pairs. It provides quick access to values based on their keys. It uses a technique called hashing to efficiently organise and retrieve data. This implementation ensures fast data storage and retrieval, making it a popular choice for various applications in Java programming.

It is an important data structure and its implementation is provided in Java. But, in this article, we will see the hashmap internal implementation.

What is a HashMap In Java?

A map contains key and value pairs. Java has the Map interface, and HashMap is a hashtable-based implementation of this interface. The HashMap class of Java is almost the same as Hashtable, except that it is unsynchronized and permits nulls.

HashMap provides constant-time performance for basic operations such as insertion and deletion.

Before understanding the internal working of HashMap, you will first understand the hashCode() and equals() methods.

- **equals()**
  The equals() method is used to compare two objects. It returns true if the objects are equal. By default, it checks if the memory addresses of the two objects are the same. We override the equals() method to provide custom logic. The overridden equals() should follow some standard properties:
    - Reflexive: An object must be equal to itself.
    - Symmetric: If the first_object is equal to the second_object, then

the second_object must be equal to the first_object.

- Transitive: If the first_object equals the second_object and the second_object equals the third_object. Then the first_object must be equal to the third_object.

- Comparison with null: equals() should return false if compared to a null object, and it should handle null parameter checks.

- **hashCode()**
  The hashCode() method returns the hash code of an object. By default, it returns the memory reference of an object in integer form. Definition of the hashCode() method is public native hashCode(). It indicates the implementation of hashCode() is native because Java has no direct method to fetch the object's reference. You can also override the default hashCode() method.

  In HashMap, hashCode() is used to calculate the bucket and therefore calculate the index.
  Note: The hash code of the null key is 0.

- **Buckets:**
  In a HashMap, the array is divided into multiple buckets used to store nodes. A bucket can hold numerous nodes forming a linked list or a balanced tree in case of collisions. Buckets are different in capacity. The relation between bucket and capacity is shown below:

Depending on the hashCode() method, a bucket can have multiple nodes.

Scenarios to Use HashMap in Java Applications

A HashMap in Java is ideal for situations where fast lookup, insertion, and deletion operations are required. Below are some common scenarios where using a HashMap is beneficial:

- **Storing and Retrieving Data by Unique Keys**: When you need to associate a unique identifier (like an ID or username) with a value (like user details), HashMap allows quick access using that key.
- **Caching Results**: HashMaps are often used in caching mechanisms to store already-computed results and retrieve them quickly, improving performance.
- **Implementing Lookup Tables**: Useful for storing mappings like country codes to country names, error codes to messages, or file extensions to MIME types.
- **Counting Frequency of Elements**: Frequently used in algorithms to count how often an item appears (e.g., words in a sentence or votes in an election system).
- **Configuration Settings or Parameters**: When you have a dynamic set of key-value configuration properties, a HashMap makes it easy to manage and access them.
- **Building Graph Structures**: HashMaps

can be used to store adjacency lists in graph representations, with nodes as keys and their neighbors as values.

How does HashMap implementation work in Java?

In Java, HashMap is implemented using an array of buckets, where each bucket is essentially a linked list of key-value pairs. The process works as follows:

- **Hashing:** When you insert a key-value pair into the HashMap, Java computes the hash code of the key using the **hashCode()** method. The hash code is then used to determine the index (bucket) where the key-value pair will be stored in the array.
- **Bucket Indexing:** The hash code is used to calculate the index of the array where the key-value pair will be stored. Java applies a hash function to the hash code to ensure that it maps to a valid index in the array.
- **Collision Handling:** Since multiple keys may hash to the same index, collisions can occur. In the event of a collision, the key-value pairs are stored in the same bucket as a linked list. This means that multiple key-value pairs can exist within the same bucket.
- **Retrieval:** When you want to retrieve a value associated with a key, Java calculates the hash code of the key, determines the bucket index, and then traverses the linked list (if any) in that bucket to find the key-value pair with the matching key.
- **Resize:** As elements are added to the HashMap, it may reach a certain threshold (load factor) where it needs to

be resized to maintain performance. When resizing occurs, the array of buckets is expanded, and all key-value pairs are rehashed and redistributed into the new array based on their updated hash codes.

Insert Key, Value pair in HashMap

The HashMap in Java is a data structure that stores entries in key-value pairs, and we can access them using an index of another type. It means a corresponding value is stored in a hashmap for every unique key. You can access the value using the key. If you try inserting a duplicate key, the new value will replace the previous one for the specific key. The keys need to be unique, whereas values need not be unique. Let's see some examples of how to add these pairs and how we can access the values using the corresponding keys.

Example

Now, we will write a program that shows how to create a hashmap and insert key-value pairs.

Code:

Java

```
import java.util.HashMap;

public class CodingNinjas {
    public static void main(String[] args)
{
        // Create a HashMap
        HashMap<String, String> hashMa
p = new HashMap<>();

        // Insert key-value pair
        hashMap.put("Kanishk", "Writer");
        hashMap.put("Ayush", "Writer");
```

```
        hashMap.put("Muskan", "Writer");
        hashMap.put("Shivani", "Writer");

        System.out.println(hashMap);

        String Kanishk_Pos = hashMap.ge
t("Kanishk");
        System.out.println("Kanishk's posit
ion is: " + Kanishk_Pos);

    }
}
```


Output:

```
{Shivani=Writer, Muskan=Writer, Kanis
hk=Writer, Ayush=Writer}
Kanishk's position is: Writer
```

In the above example, we created a class named **'CodingNinjas.**' Under **void main(),** we created a HashMap named **'hashMap'** with String-String pair as key-value pair. We inserted key-value pairs in our hashmap using the **'put()'** function respectively. Now it does two actions. First, it prints the whole hashmap, and later, using the **'get()'** function, it finds the key **'Kanishk'** value and stores it in a variable named **'Kanishk_Pos.'** Then it displays the value stored in the variable.

Calculating Index

With the help of the hashCode() function, we calculate the index. The following are the steps you need to follow to calculate the index:

1. Calculate the hash code for the key to calculate the index using the

hashCode() method.

2. After calculating the hash code, use the modulo operator (%) to obtain the remainder when dividing the transformed hash code by the size of the array. This ensures that the resulting index is within the valid range of bucket indices.

Once the index is calculated, use it to store the key-value pair, or you can also use it to retrieve the pair. The following code will help you understand better.

Code:

Java

```
import java.util.HashMap;
import java.util.Map;

public class CodingNinjas {
   public static void main(String[] args)
{
      HashMap<String, String> hashMa
p = new HashMap<>();

      hashMap.put("Kanishk", "Writer");
      hashMap.put("Ayush", "Writer");
      hashMap.put("Muskan", "Writer");
      hashMap.put("Shivani", "Writer");

      for (Map.Entry<String, String> entr
y : hashMap.entrySet()) {
         String key = entry.getKey();
         int hashCode = key.hashCode();
         int index = hashCode % hashM
ap.size();
         System.out.println("Key: " + key
+ ", Index: " + index);
      }
   }
}
```

You can also try this code with Online Java Compiler

Run Code

Output:

```
Key: Muskan, Index: 0
Key: Ayush, Index: 2
Key: Kanishk, Index: 2
Key: Shivani, Index: 3
```

Handling hash collisions in Java

“A collision in a HashMap is a state in which two or more keys produce the same hash value and point to the same bucket or index.” Let's understand it with the help of an example.

Code:

Java

```
import java.util.HashMap;
import java.util.Map;

public class CodingNinjas {
    private String name;

    public CodingNinjas(String name) {
        this.name = name;
    }

    @Override
    public int hashCode() {
        int result = 0;
        for (int i = 0; i < name.length(); i++)
{
            result += name.charAt(i);
        }
        return result;
    }

    public static void main(String[] args)
{

        HashMap<CodingNinjas, String>
hashMap = new HashMap<>();
```

```
        hashMap.put(new CodingNinjas
("kanishk"), "Writer");
        hashMap.put(new CodingNinjas
("Muskan"), "Writer");
        hashMap.put(new CodingNinjas
("vaibhvi"), "Writer");

        for (Map.Entry<CodingNinjas, Stri
ng> entry : hashMap.entrySet()) {
            CodingNinjas key = entry.getKey
();
            int hashCode = key.hashCode();
            int index = (hashCode % hashM
ap.size());
            System.out.println("Key: " + key.
name + ", Index: " + index + ", Hash Co
de: " + hashCode);
        }
    }
}
```


Output:

```
Key: kanishk, Index: 1, Hash Code: 745
Key: vaibhvi, Index: 1, Hash Code: 745
Key: Muskan, Index: 2, Hash Code: 623
```

As you can see in the above output, **"kanishk"** and **"vaibhvi"** have the same hash value and point to the same index, i.e., 1. This is a case of hash collision.

In cases like these, we check via equals() if the two keys are the same; if the keys are the same, then replace the value with the new value. Otherwise, connect the node objects via a linked list; both are stored at the same index. Before discussing the new HashMap, we should know the structure of a node.

```
{
 Hash code = ....
 Key =...
 Value =...
```

```
 Next Node =...
}
```

Now the hashmap for the above example becomes:

get() method in HashMap

We already saw the usage of the **'get()'** method earlier in this blog, but here we will discuss it in detail.

The **'get()'** retrieves the value for a particular key from the HashMap. The syntax for it is as follows:

```
Hashmap_name.get(key_name);
```

Now let's discuss how the **'get()'** method gets the value.

1. Calculate the hash code of the key.
2. Calculate the index by using the index method.
3. Go to index and compare the first element's key with the given key. If both are equals, return the value; otherwise, check for the next element if it exists.
4. If the next node is null, then return null.
5. If the next node is not null, traverse to the second element and repeat process three until the key is not found or the next is not null.

Handling resizing in Java

When the number of elements in a HashMap reaches a certain threshold (determined by its load factor), resizing occurs to maintain performance. Resizing involves creating a new array of buckets (typically double the size of the original array) and rehashing all existing key-value pairs into the new array. This process ensures that the elements are evenly distributed across the array, reducing collisions and maintaining the efficiency of the HashMap.

Adding key-value pairs to the HashMap in Java

When adding a key-value pair to a HashMap in Java, the following steps are typically followed:

1. Compute the hash code of the key using the **hashCode()** method.
2. Use the hash code to determine the index (bucket) where the key-value pair will be stored in the array.
3. If the bucket is empty, add the key-value pair directly.
4. If the bucket is not empty due to a collision, add the key-value pair to the end of the linked list in that bucket.

Retrieving values

from the HashMap

To retrieve a value associated with a key from a HashMap in Java:

1. Compute the hash code of the key using the **hashCode()** method.
2. Use the hash code to determine the index (bucket) where the key-value pair is stored in the array.
3. Traverse the linked list (if any) in that bucket to find the key-value pair with the matching key.
4. Return the value associated with the key if found, otherwise return **null** to indicate that the key is not present in the HashMap.

Removing key-value pairs from the HashMap

To remove a key-value pair from a HashMap in Java:

1. Compute the hash code of the key using the **hashCode()** method.
2. Use the hash code to determine the index (bucket) where the key-value pair is stored in the array.
3. Traverse the linked list (if any) in that bucket to find the key-value pair with the matching key.
4. If the key is found, remove the key-value pair from the linked list.
5. If the linked list becomes empty after removing the key-value pair, set the bucket to null to free up memory.

Methods in Java HashMap

Method	Description
put(K key, V value)	Inserts a key-value pair into the map. If the key already exists, the value is updated.
get(Object key)	Returns the value to which the specified key is mapped, or null if the map contains no mapping for the key.
remove(Object key)	Removes the mapping for the specified key from this map if present.
containsKey(Object key)	Returns true if this map contains a mapping for the specified key.
containsValue(Object value)	Returns true if this map maps one or more keys to the specified value.
size()	Returns the number of key-value mappings in this map.
isEmpty()	Returns true if this map contains no

Method	Description
	key-value mappings.
clear()	Removes all of the mappings from this map.
putAll(Map<? extends K, ? extends V> m)	Copies all of the mappings from the specified map to this map.
keySet()	Returns a Set view of the keys contained in this map.
values()	Returns a Collection view of the values contained in this map.
entrySet()	Returns a Set view of the key-value mappings contained in this map.
putIfAbsent(K key, V value)	If the specified key is not already associated with a value, associates it with the given value.
replace(K key, V value)	Replaces the entry for the specified key only if it is currently mapped to some value.

Method	Description
replace(K key, V oldValue, V newValue)	Replaces the entry for the specified key only if it is currently mapped to the specified value.
computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)	If the specified key is not already associated with a value, attempts to compute its value using the given mapping function and enters it into the map.
computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)	If the value for the specified key is present, attempts to compute a new mapping for the key using the given remapping function.
merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)	If the specified key is not already associated with a value, associates it with the given value. Otherwise, merges the new value with the existing value using the

Method	Description
	given remapping function.

Features of a Java HashMap

- **Key-Value Pair Storage**: HashMap stores data in the form of key-value pairs, allowing fast retrieval using the key.
- **Allows One Null Key and Multiple Null Values**: HashMap permits one null key and any number of null values.
- **No Duplicate Keys**: Keys must be unique. If a duplicate key is added, it overwrites the existing value.
- **Unordered**: HashMap does not maintain the order of insertion. The entries are not stored in any specific sequence.
- **Non-Synchronized**: HashMap is not thread-safe by default. If used in a multi-threaded environment, external synchronization is required.
- **Efficient Performance**: Provides constant-time performance (O(1)) for basic operations like get() and put() under average conditions.
- **Implements Map Interface**: HashMap is part of the Java Collections Framework and implements the Map interface.
- **Uses Hashing**: Internally uses a hash table and hashing mechanism to store and retrieve values efficiently.

Frequently Asked Questions

How hashCode() and equals() are used in HashMap?

hashCode() is used to determine the bucket for storing a key-value pair, while equals() ensures that keys are compared correctly within that bucket, allowing for retrieval or replacement of entries.

What is the difference between HashMap and LinkedHashMap?

HashMap does not maintain any order of keys, while LinkedHashMap maintains the insertion order or access order of keys. This makes LinkedHashMap ideal when predictable iteration order is needed.

Can a HashMap store duplicate keys?

No, a HashMap cannot store duplicate keys. If a duplicate key is inserted, the existing value is replaced with the new one.

Why is HashMap used?

HashMap is used for fast data retrieval and efficient key-value storage where quick lookups, insertions, and deletions are needed based on unique keys.

What is the time complexity of basic operations in a HashMap?

The time complexity for basic operations like get(), put(), and remove() in a HashMap is O(1) on average, but it can degrade to O(n) in worst-case scenarios due to hash collisions.

How is HashMap implemented in Java 8?

Java 8 HashMaps use an array of buckets,

but they optimize with tree structures for better performance. Additionally, Java 8 introduces the concept of balanced trees to handle collision resolution more efficiently.

Conclusion

HashMap in Java is a powerful data structure used for storing key-value pairs with efficient performance. It is widely used for quick lookups, caching, and mapping scenarios. With its flexible design and constant-time operations, Java HashMap is essential for developers handling dynamic data. Understanding HashMap implementation in Java helps in writing optimized and scalable applications.

Recommended Articles

- Pair sum, Check duplicate
- Count distinct substrings
- Maximum frequency number

Previous article

Difference between HashMap and...

Next article

Difference Between HashSet and...

Related articles

LRU Cache Implementation

Binary Subarrays with Sum

Double Hashing

Rehashing in Data Structure

HashMap in Java

Library:

Java • Python • C Programming Language • C++ Programming Language • Cloud Computing • Node JS • Machine Learning • Deep Learning • Big Data • Operating System • Go Language • C# • Ruby • Amazon Web Services • Microsoft Azure • Google Cloud Platform • Data Warehousing • Internet of Things

Contact us

1800-123-3598

code360@codingninjas.com

Privacy policy

Terms & conditions

Download the naukri app