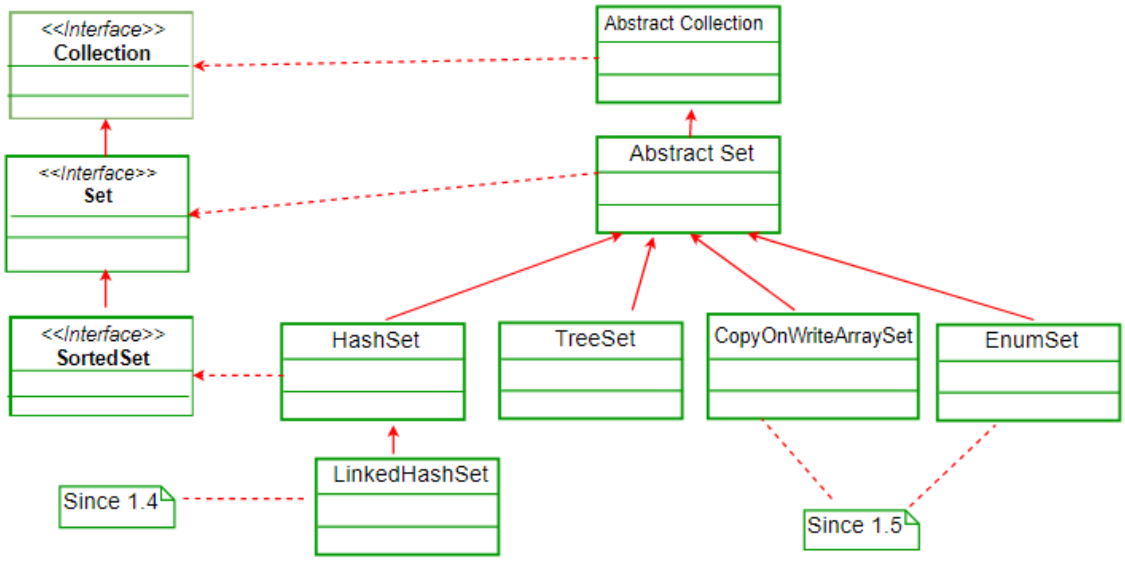
HashSet in Java

The HashSet class implements the Set interface, backed by a hash table which is actually a HashMap instance. No guarantee is made as to the iteration order of the set which means that the class does not guarantee the constant order of elements over time. This class permits the null element. The class also offers constant time performance for the basic operations like add, remove, contains and size assuming the hash function disperses the elements properly among the buckets, which we shall see further in the article.

Few important features of HashSet are:

- Implements Set Interface.
- Underlying data structure for HashSet is hashtable.
- As it implements the Set Interface, duplicate values are not allowed.
- Objects that you insert in HashSet are not guaranteed to be inserted in same order. Objects are inserted based on their hash code.
- NULL elements are allowed in HashSet.
- HashSet also implements Searlizable and Cloneable interfaces.



Now for the maintenance of constant time performance, iterating over HashSet requires time proportional to the sum of the HashSet instance's size (the number of elements) plus the "capacity" of the backing HashMap instance (the number of buckets). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

Initial Capacity: The initial capacity means the number of buckets when hashtable (HashSet internally uses hashtable data structure) is created. The number of buckets will be automatically increased if the current size gets full.
Load Factor: The load factor is a measure of how full the HashSet is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

$$\text{load factor} = \frac{\text{Number of stored elements in the table}}{\text{Size of the hash table}}$$

Example: If internal capacity is 16 and load factor is 0.75 then, number of buckets will automatically get increased when the table has 12 elements in it.

Effect on performance:
Load factor and initial capacity are two main factors that affect the performance of HashSet operations. Load factor of 0.75 provides very effective performance as respect to time and space complexity. If we increase the load factor value more than that then memory overhead will be reduced (because it will decrease internal rebuilding operation) but, it will affect the add and search operation in hashtable. To reduce the rehashing operation we should choose initial capacity wisely. If initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operation will ever occur.

NOTE: The implementation in a HashSet is not synchronized, in the sense that if multiple threads access a hash set concurrently, and at least one of the threads modifies the set, it must be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be "wrapped" using the Collections.synchronizedSet method.

This is best done at creation time, to prevent accidental unsynchronized access to the set as shown below:

```
Set s = Collections.synchronizedSet(new HashSet(...));
```

Constructors in HashSet:

1. **HashSet h = new HashSet();**
   Default initial capacity is 16 and default load factor is 0.75.
2. **HashSet h = new HashSet(int initialCapacity);**
   default loadFactor of 0.75
3. **HashSet h = new HashSet(int initialCapacity, float loadFactor**);
4. **HashSet h = new HashSet(Collection C);**

Below program illustrates few basic operations of HashSet:

```
// Java program to demonstrate working of HashSet
import java.util.*;

class Test
{
    public static void main(String[]args)
    {
        HashSet<String> h = new HashSet<String>();

        // Adding elements into HashSet usind add()
        h.add("India");
        h.add("Australia");
        h.add("South Africa");
        h.add("India");// adding duplicate elements

        // Displaying the HashSet
        System.out.println(h);
        System.out.println("List contains India or not:" +
                           h.contains("India"));

        // Removing items from HashSet using remove()
        h.remove("Australia");
        System.out.println("List after removing Australia:"+h);

        // Iterating over hash set items
        System.out.println("Iterating over list:");
        Iterator<String> i = h.iterator();
        while (i.hasNext())
            System.out.println(i.next());
    }
}
```

Output:

```
[South Africa, Australia, India]
List contains India or not:true
List after removing Australia:[South Africa, India]
Iterating over list:
South Africa
India
```

Internal working of a HashSet

All the classes of Set interface internally backed up by Map. HashSet uses HashMap for storing its object internally. You must be wondering that to enter a value in HashMap we need a key-value pair, but in HashSet we are passing only one value.

Storage in HashMap

Actually the value we insert in HashSet acts as key to the map Object and for its value java uses a constant variable. So in key-value pair all the keys will have same value.

Implementation of HashSet in java doc:

```
private transient HashMap map;

// Constructor - 1
// All the constructors are internally creating HashMap Object.
public HashSet()
{
    // Creating internally backing HashMap object
    map = new HashMap();
}

// Constructor - 2
public HashSet(int initialCapacity)
{
    // Creating internally backing HashMap object
    map = new HashMap(initialCapacity);
}

// Dummy value to associate with an Object in Map
private static final Object PRESENT = new Object();
```

If we look at add() method of HashSet class:

```
public boolean add(E e)
{
   return map.put(e, PRESENT) == null;
}
```

We can notice that, add() method of HashSet class internally calls put() method of backing HashMap object by passing the element you have specified as a key and constant “PRESENT” as its value.

remove() method also works in the same manner. It internally calls remove method of Map interface.

```
public boolean remove(Object o)
{
  return map.remove(o) == PRESENT;
}
```

Time Complexity of HashSet Operations: The underlying data structure for HashSet is hashtable. So amortize (average or usual case) time complexity for add, remove and look-up (contains method) operation of HashSet takes **O(1)** time.

Methods in HashSet:

1. boolean add(E e): Used to add the specified element if it is not present, if it is present then return false.
2. void clear(): Used to remove all the elements from set.
3. boolean contains(Object o): Used to return true if an element is present in set.
4. boolean remove(Object o): Used to remove the element if it is present in set.
5. Iterator iterator(): Used to return an iterator over the element in the set.
6. boolean isEmpty(): Used to check whether the set is empty or not. Returns true for empty and false for non-empty condition for set.
7. int size(): Used to return the size of the set.
8. Object clone(): Used to create a shallow copy of the set.

Reference: https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html

This article is contributed by **Dharmesh Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Recommended Posts:

Traverse through a HashSet in Java

HashSet vs TreeSet in Java

How to sort HashSet in Java

HashSet contains() Method in Java