**HashSet**

The **HashSet** class implements the [Set interface](https://www.geeksforgeeks.org/set-in-java/), backed by a hash table which is actually a [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/) instance. 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.

**Features of HashSet:**

* Implements [Set Interface](https://www.geeksforgeeks.org/set-in-java/).
* The underlying data structure for HashSet is [Hashtable](https://www.geeksforgeeks.org/hashtable-in-java/).
* As it implements the Set Interface, duplicate values are not allowed.
* Objects that you insert in HashSet are not guaranteed to be inserted in the same order. Objects are inserted based on their hash code.
* NULL elements are allowed in HashSet.
* HashSetalso implements **Serializable** and **Cloneable** interfaces.

Application

Description automatically generated**Hierarchy**

HashSet extends [Abstract Set<E>](https://www.geeksforgeeks.org/abstractset-class-in-java-with-examples/#:~:text=The%20AbstractSet%20class%20in%20Java,implementation%20of%20the%20Set%20interface.) class and implements [Set<E>](https://www.geeksforgeeks.org/set-in-java/), [Cloneable](https://www.geeksforgeeks.org/marker-interface-java/#:~:text=Cloneable%20interface%20%3A%20Cloneable%20interface%20is,of%20instances%20of%20that%20class.), and [Serializable](https://www.geeksforgeeks.org/serialization-in-java/) interfaces

where E is the type of elements maintained by this set. The directly known subclass of HashSet is [LinkedHashSet](https://www.geeksforgeeks.org/linkedhashset-in-java-with-examples/).

* **Initial Capacity:** The initial capacity means the number of buckets when hashtable is created. The number of buckets will be automatically increased if the current size gets full.

Number of stored elements in the table

* **Load Factor** = ------------------------------------------------------

                         Size of the hash table

* **Effect on performance:**  Load factor and initial capacity are two main factors that affect the performance of HashSet operations.
* **Internal working of a HashSet:**All the classes of Set interface are 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 a key to the map Object and for its value, java uses a constant variable. So in the key-value pair, all the values will be the same.

**Constructors of HashSet class**

**1. HashSet()**: This constructor is used to build an empty HashSet object in which the default initial capacity is 16 and the default load factor is 0.75. If we wish to create an empty HashSet with the name hs, then, it can be created as:

HashSet<E> hs = new HashSet<E>();

**2. HashSet(int initialCapacity):** This constructor is used to build an empty HashSet object in which the initialCapacity is specified at the time of object creation. Here, the default loadFactor remains 0.75.

HashSet<E> hs = new HashSet<E>(int initialCapacity);

**3. HashSet(int initialCapacity, float loadFactor):** This constructor is used to build an empty HashSet object in which the initialCapacity and loadFactor are specified at the time of object creation.

HashSet<E> hs = new HashSet<E>(int initialCapacity, float loadFactor);

**4. HashSet(Collection):** This constructor is used to build a HashSet object containing all the elements from the given collection. In short, this constructor is used when any conversion is needed from any Collection object to the HashSet object. If we wish to create a HashSet with the name hs, it can be created as:

HashSet<E> hs = new HashSet<E>(Collection C);

**Methods in HashSet**

| **METHOD** | **DESCRIPTION** |
| --- | --- |
| [add(E e)](https://www.geeksforgeeks.org/hashset-add-method-in-java/) | Used to add the specified element if it is not present, if it is present then return false. |
| [clear()](https://www.geeksforgeeks.org/hashset-clear-method-in-java/) | Used to remove all the elements from set. |
| [contains(Object o)](https://www.geeksforgeeks.org/hashset-contains-method-in-java/) | Used to return true if an element is present in set. |
| [remove(Object o)](https://www.geeksforgeeks.org/hashset-remove-method-in-java/) | Used to remove the element if it is present in set. |
| [iterator()](https://www.geeksforgeeks.org/hashset-iterator-method-in-java/) | Used to return an iterator over the element in the set. |
| [isEmpty()](https://www.geeksforgeeks.org/hashset-isempty-method-in-java/) | Used to check whether the set is empty or not. Returns true for empty and false for a non-empty condition for set. |
| [size()](https://www.geeksforgeeks.org/hashset-size-method-in-java/) | Used to return the size of the set. |
| [clone()](https://www.geeksforgeeks.org/hashset-clone-method-in-java/) | Used to create a shallow copy of the set. |

**LinkedHashSet**

Java LinkedHashSet class is a Hashtable and Linked list implementation of the Set interface. It inherits the HashSet class and implements the Set interface.

The important points about the Java LinkedHashSet class are:

* Java LinkedHashSet class contains unique elements only like HashSet.
* Java LinkedHashSet class provides all optional set operations and permits null elements.
* Java LinkedHashSet class is non-synchronized.
* Java LinkedHashSet class maintains insertion order.

### LinkedHashSet Class Declaration

* publicclass LinkedHashSet<E> extends HashSet<E> implements Set<E>, Cloneable, Serializable
* **All Implemented Interfaces are as listed below:**
* Serializable
* Cloneable,
* Iterable<E>
* Collection<E>
* Set<E>

### Constructors of Linked HashSet Class

* **1. LinkedHashSet():** This constructor is used to create a default HashSet
* LinkedHashSet<E> hs = new LinkedHashSet<E>();
* **2. Linked HashSet(Collection C):** Used in initializing the HashSet with the elements of the collection C.
* LinkedHashSet<E> hs = new LinkedHashSet<E>(Collection c);
* **3. Linked HashSet(int size):** Used to initialize the size of the LinkedHashSet with the integer mentioned in the parameter.
* LinkedHashSet<E> hs = new LinkedHashSet<E>(int size);

**4. LinkedHashSet(int capacity, float fillRatio):** Can be used to initialize both the capacity and the fill ratio, also called the load capacity of the LinkedHashSet with the arguments mentioned in the parameter. When the number of elements exceeds the capacity of the hash set is multiplied with the fill ratio thus expanding the capacity of the LinkedHashSet.

LinkedHashSet<E> hs = new LinkedHashSet<E>(int capacity, int fillRatio);

**When to use?**

**•**It contains only a unique element like HashSet

•It maintains an insertion order

•Whenever we want to store the unique elements with their insertion order, then we can use LinkedHashSet

**Insert Elements to LinkedHashSet**

* add() - inserts the specified element to the linked hash set
* addAll() - inserts all the elements of the specified collection to the linked hash set

**​**

| **METHOD** | **DESCRIPTION** |
| --- | --- |
| [equals(Object o)](https://www.geeksforgeeks.org/abstractset-equals-method-in-java-with-examples/?ref=rp) | Compares the specified object with this set for equality. |
| [hashCode()](https://www.geeksforgeeks.org/abstractset-hashcode-method-in-java-with-examples/?ref=rp) | Returns the hash code value for this set. |
| [removeAll(Collection c)](https://www.geeksforgeeks.org/abstractset-removeall-method-in-java-with-examples/?ref=rp) | Removes from this set all of its elements that are contained in the specified collection (optional operation). |

|  |  |
| --- | --- |
| [addAll(Collection<? extends E> c)](https://www.geeksforgeeks.org/abstractcollection-addall-method-in-java-with-examples/) | Adds all of the elements in the specified collection to this collection (optional operation). |
| [containsAll(Collection<?> c)](https://www.geeksforgeeks.org/abstractcollection-containsall-method-in-java-with-examples/) | Returns true if this collection contains all of the elements in the specified collection. |
| [retainAll(Collection<?> c)](https://www.geeksforgeeks.org/abstractcollection-retainall-method-in-java-with-examples/) | Retains only the elements in this collection that are contained in the specified collection (optional operation). |
| [toArray()](https://www.geeksforgeeks.org/abstractcollection-toarray-method-in-java-with-examples/) | Returns an array containing all of the elements in this collection. |
| [toArray(T[] a)](https://www.geeksforgeeks.org/abstractcollection-toarray-method-in-java-with-examples/) | Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array. |
| [toString()](https://www.geeksforgeeks.org/abstractcollection-tostring-method-in-java-with-examples/) | Returns a string representation of this collection. |
| parallelStream() | Returns a possibly parallel Stream with this collection as its source. |
| removeIf(Predicate<? super E> filter) | Removes all of the elements of this collection that satisfy the given predicate. |
| stream() | Returns a sequential Stream with this collection as its source. |
| [add(E e)](https://www.geeksforgeeks.org/hashset-add-method-in-java/) | Adds the specified element to this set if it is not already present. |
| [clear()](https://www.geeksforgeeks.org/hashset-clear-method-in-java/) | Removes all of the elements from this set. |
| [clone()](https://www.geeksforgeeks.org/hashset-clone-method-in-java/) | Returns a shallow copy of this HashSet instance: the elements themselves are not cloned. |
| [contains(Object o)](https://www.geeksforgeeks.org/hashset-contains-method-in-java/) | Returns true if this set contains the specified element. |
| [isEmpty()](https://www.geeksforgeeks.org/hashset-isempty-method-in-java/) | Returns true if this set contains no elements. |
| [iterator()](https://www.geeksforgeeks.org/hashset-iterator-method-in-java/) | Returns an iterator over the elements in this set. |
| [remove(Object o)](https://www.geeksforgeeks.org/hashset-remove-method-in-java/) | Removes the specified element from this set if it is present. |
| [size()](https://www.geeksforgeeks.org/hashset-size-method-in-java/) | Returns the number of elements in this set (its cardinality). |

**COMPARISON BETWEEN HASHSET AND LINKEDHASHSET**

​

|  |  |  |
| --- | --- | --- |
| **Property** | **HashSet** | **LinkedHashSet** |
| Data structure | Uses Hashtable | Uses HashTable and doubly linked list |
| Technique to store the elements | Hashing | Hashing |
| Declaration | HashSet obj = new HashSet(); | LinkedHashSet obj = new LinkedHashSet(); |
| Implements | Set interface | Set interface |
| Insertion Order | Does not provide any insertion order | Provides an insertion order |
| Null elements | Only one null element | Only one null element |
|  |  |  |

**TREESET**

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.

* Java TreeSet class contains unique elements only like HashSet.
* Java TreeSet class access and retrieval times are quiet fast.
* Java TreeSet class doesn't allow null element.
* Java TreeSet class is non synchronized.
* Java TreeSet class maintains ascending order.
* Java TreeSet class contains unique elements only like HashSet.
* Java TreeSet class access and retrieval times are quite fast.
* Java TreeSet class doesn't allow null elements.
* Java TreeSet class is non-synchronized.
* Java TreeSet class maintains ascending order.

**INTERNAL WORKING**

TreeSet is being implemented using a binary search tree, which is self-balancing just like a Red-Black Tree. Therefore, operations such as a search, remove, and add consume O(log(N)) time. The reason behind this is there in the self-balancing tree. It is there to ensure that the tree height never exceeds O(log(N)) for all of the mentioned operations. Therefore, it is one of the efficient data structures in order to keep the large data that is sorted and also to do operations on it.

**SYNCRONIZATION**

As already mentioned above, the TreeSet class is not synchronized. It means if more than one thread concurrently accesses a tree set, and one of the accessing threads modify it, then the synchronization must be done manually. It is usually done by doing some object synchronization that encapsulates the set. However, in the case where no such object is found, then the set must be wrapped with the help of the Collections.synchronizedSet() method. It is advised to use the method during creation time in order to avoid the unsynchronized access of the set. The following code snippet shows the same.

*TreeSet treeSet = new TreeSet();*

*Set syncrSet = Collections.synchronziedSet(treeSet);*

**Application

Description automatically generated with low confidenceHIERARCHY**

**DECLARATION**

*public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>, Cloneable, Serializable*

**CONSTRUCTORS**

1. TreeSet()

It is used to construct an empty tree set that will be sorted in ascending order according to the natural order of the tree set.

1. TreeSet(Collection<? extends E> c)

It is used to build a new tree set that contains the elements of the collection c.

1. TreeSet(Comparator<? super E> comparator)

It is used to construct an empty tree set that will be sorted according to given comparator.

1. TreeSet(SortedSet<E> s)

It is used to build a TreeSet that contains the elements of the given SortedSet.

**METHODS OF TREESET**

* add(Object o) : This method will add the specified element according to the same sorting order mentioned during the creation of the TreeSet. Duplicate entries will not get added.
* addAll(Collection c): This method will add all elements of the specified Collection to the set. Elements in the Collection should be homogeneous otherwise ClassCastException will be thrown. Duplicate Entries of Collection will not be added to TreeSet.
* clear(): This method will remove all the elements.
* clone(): The method is used to return a shallow copy of the set, which is just a simple copied set.
* contains(): This method will return true if a given element is present in TreeSet else it will return false.
* descendingIterator(): This method returns an iterator over the elements in this set in descending order.
* isEmpty(): This method is used to return true if this set contains no elements or is empty and false for the opposite case.
* last(): This method will return the last element in TreeSet if TreeSet is not null else it will throw NoSuchElementException.
* size(): This method is used to return the size of the set or the number of elements present in the set.

**SAMPLE PROGRAMS**

1. **SAMPLE WORKING OF TREESET**

import java.util.\*;

class GFG {

public static void main(String[] args)

{

Set<String> ts1 = new TreeSet<>();

ts1.add("A");

ts1.add("B");

ts1.add("C");

ts1.add("C");

System.out.println(ts1);

}

}

OUTPUT:

Ajay

Ravi

Vijay

import java.util.\*;

class TreeSet2{

public static void main(String args[]){

TreeSet<String> set=new TreeSet<String>();

set.add("Ravi");

set.add("Vijay");

set.add("Ajay");

System.out.println("Traversing element through Iterator in descending order");

Iterator i=set.descendingIterator();

while(i.hasNext())

{

System.out.println(i.next());

}

}

}

OUTPUT

Traversing element through Iterator in descending order

Vijay

Ravi

Ajay

Traversing element through NavigableSet in descending order

Vijay

Ravi

Ajay

import java.util.\*;

class TreeSet3{

public static void main(String args[]){

TreeSet<Integer> set=new TreeSet<Integer>();

set.add(24);

set.add(66);

set.add(12);

set.add(15);

System.out.println("Lowest Value: "+set.pollFirst());

System.out.println("Highest Value: "+set.pollLast());

}

}

OUTPUT

Lowest Value: 12

Highest Value: 66