Difference between TreeSet, LinkedHashSet and HashSet in Java

TreeSet, LinkedHashSet and HashSet in Java are three Set implementation in collection framework and like many others they are also used to store objects. Main feature of TreeSet is sorting,  LinkedHashSet is insertion order and HashSet is just general purpose collection for storing object. HashSet is implemented using HashMap in Java while TreeSet is implemented using TreeMap.  TreeSet is a SortedSetimplementation which allows it to keep elements in the sorted order defined by either Comparable or Comparator interface. Comparable is used for natural order sorting and Comparator for custom order sorting of objects, which can be provided while creating instance of TreeSet. Anyway before seeing difference between TreeSet, LinkedHashSet and HashSet, let's see some similarities between them:

1) **Duplicates** : All three implements Set interface means they are not allowed to store duplicates.

2) **Thread safety**: HashSet, TreeSet and LinkedHashSet are not thread-safe, if you use them in multi-threading environment where at least one Thread  modifies Set you need to externally synchronize them.

3) **Fail-Fast Iterator** : Iterator returned by TreeSet, LinkedHashSet and HashSet are fail-fast Iterator. i.e. If Iterator is modified after its creation by any way other than Iterators remove() method, it will throw ConcurrentModificationException with best of effort.

**Difference between HashSet, LinkedHashSet and TreeSet in Java** :

**Performance** and **Speed** : First difference between them comes in terms of  speed.  HashSet is fastest, LinkedHashSet is second on performance or almost similar to HashSet but TreeSet is bit slower because of sorting operation it needs to perform on each insertion. TreeSet provides guaranteed O(log(n)) time for common operations like add, remove and contains, while HashSet and LinkedHashSet offer constant time performance e.g. O(1) for add, contains and remove given hash function uniformly distribute elements in bucket.

**Ordering :** HashSet does not maintain any order while LinkedHashSet maintains insertion order of elements much like List interface and TreeSet maintains sorting order or elements.

**Internal Implementation :** HashSet is backed by an HashMap instance, LinkedHashSet is implemented using HashSet and LinkedList while TreeSet is backed up by NavigableMap in Java and by default it uses TreeMap.

**null** : Both HashSet and LinkedHashSet allows null but TreeSet doesn't allow null but TreeSet doesn't allow null and throw java.lang.NullPointerException when you will insert null into TreeSet. Since TreeSet uses compareTo() method of respective elements to compare them  which throws NullPointerException while comparing with null, here is an example:

TreeSet cities

Exception in thread "main" java.lang.NullPointerException

        at java.lang.String.compareTo(String.java:1167)

        at java.lang.String.compareTo(String.java:92)

        at java.util.TreeMap.put(TreeMap.java:545)

        at java.util.TreeSet.add(TreeSet.java:238)

**Comparison :** HashSet and LinkedHashSet uses equals() method in Java for comparison but TreeSet uses compareTo() method for maintaining ordering. That's why compareTo() should be consistent to equals in Java. failing to do so break general contact of Set interface i.e. it can permit duplicates.

## Differences Between HashSet, LinkedHashSet and TreeSet In Java :

|  |  |  |  |
| --- | --- | --- | --- |
|  | **HashSet** | **LinkedHashSet** | **TreeSet** |
| How they work internally? | HashSet uses HashMap internally to store it’s elements. | LinkedHashSet uses  LinkedHashMap internally to store it’s elements. | TreeSet uses TreeMap internally to store it’s elements. |
| Order Of Elements | HashSet doesn’t maintain any order of elements. | LinkedHashSet maintains insertion order of elements. i.e elements are placed as they are inserted. | TreeSet orders the elements according to supplied Comparator. If no comparator is supplied, elements will be placed in their natural ascending order. |
| Performance | HashSet gives better performance than the LinkedHashSet and TreeSet. | The performance of LinkedHashSet is between HashSet and TreeSet. It’s performance is almost similar to HashSet. But slightly in the slower side as it also maintains LinkedList internally to maintain the insertion order of elements. | TreeSet gives less performance than the HashSet and LinkedHashSet as it has to sort the elements after each insertion and removal operations. |
| Insertion, Removal And Retrieval Operations | HashSet gives performance of order O(1) for insertion, removal and retrieval operations. | LinkedHashSet also gives performance of order O(1) for insertion, removal and retrieval operations. | TreeSet gives performance of order O(log(n)) for insertion, removal and retrieval operations. |
| How they compare the elements? | HashSet uses equals() and hashCode() methods to compare the elements and thus removing the possible duplicate elements. | LinkedHashSet also uses equals() and hashCode() methods to compare the elements. | TreeSet uses compare() or compareTo() methods to compare the elements and thus removing the possible duplicate elements. It doesn’t use equals() and hashCode() methods for comparision of elements. |
| Null elements | HashSet allows maximum one null element. | LinkedHashSet also allows maximum one null element. | TreeSet doesn’t allow even a single null element. If you try to insert null element into TreeSet, it throws NullPointerException. |
| Memory Occupation | HashSet requires less memory than LinkedHashSet and TreeSet as it uses only HashMap internally to store its elements. | LinkedHashSet requires more memory than HashSet as it also maintains LinkedList along with HashMap to store its elements. | TreeSet also requires more memory than HashSet as it also maintains Comparator to sort the elements along with the TreeMap. |
| When To Use? | Use HashSet if you don’t want to maintain any order of elements. | Use LinkedHashSet if you want to maintain insertion order of elements. | Use TreeSet if you want to sort the elements according to some Comparator. |

TreeSet vs HashSet vs LinkedHashSet - Example

Let’s compare all these Set implementation on some points by writing Java program. In this example we are demonstrating difference in ordering, time taking while inserting 1M records among TreeSet, HashSet and LinkedHashSet in Java. This will help to solidify some points which discussed in earlier section and help to decide when to use HashSet, LinkedHashSet or TreeSet in Java.

**import** java.util.Arrays;  
**import** java.util.HashSet;  
**import** java.util.LinkedHashSet;  
**import** java.util.Set;  
**import** java.util.TreeSet;  
  
/\*\*  
 \* Java program to demonstrate difference between TreeSet, HashSet and LinkedHashSet

 \* in Java Collection.  
 \* @author  
 \*/  
**public** **class** SetComparision {  
    
    **public** **static** **void** main(**String** args[]){               
        **HashSet**<**String**> fruitsStore = **new** **HashSet**<**String**>();  
        **LinkedHashSet**<**String**> fruitMarket = **new** **LinkedHashSet**<**String**>();  
        **TreeSet**<**String**> fruitBuzz = **new** **TreeSet**<**String**>();  
        
        for(**String** fruit: **Arrays**.asList("mango", "apple", "banana")){  
            fruitsStore.add(fruit);  
            fruitMarket.add(fruit);  
            fruitBuzz.add(fruit);  
        }

*//no ordering in HashSet – elements stored in random order*  
        **System**.out.println("Ordering in HashSet :" + fruitsStore);

*//insertion order or elements – LinkedHashSet storeds elements as insertion*  
        **System**.err.println("Order of element in LinkedHashSet :" + fruitMarket);

*//should be sorted order – TreeSet stores element in sorted order*  
        **System**.out.println("Order of objects in TreeSet :" + fruitBuzz);

*//Performance test to insert 10M elements in HashSet, LinkedHashSet and TreeSet*  
        **Set**<**Integer**> numbers = **new** **HashSet**<**Integer**>();  
        **long** startTime = **System**.nanoTime();  
        for(**int** i =0; i<10000000; i++){  
            numbers.add(i);  
        }

        **long** endTime = **System**.nanoTime();  
        **System**.out.println("Total time to insert 10M elements in HashSet in sec : "

                            + (endTime - startTime));  
        
        
        *// LinkedHashSet performance Test – inserting 10M objects*

        numbers = **new** **LinkedHashSet**<**Integer**>();  
        startTime = **System**.nanoTime();  
        for(**int** i =0; i<10000000; i++){  
            numbers.add(i);  
        }  
        endTime = **System**.nanoTime();  
        **System**.out.println("Total time to insert 10M elements in LinkedHashSet in sec : "

                            + (endTime - startTime));

*// TreeSet performance Test – inserting 10M objects*  
        numbers = **new** **TreeSet**<**Integer**>();  
        startTime = **System**.nanoTime();  
        for(**int** i =0; i<10000000; i++){  
            numbers.add(i);  
        }  
        endTime = **System**.nanoTime();  
        **System**.out.println("Total time to insert 10M elements in TreeSet in sec : "

                            + (endTime - startTime));  
    }  
}  
  
**Output**  
Ordering in **HashSet** :[banana, apple, mango]  
Order of element in **LinkedHashSet** :[mango, apple, banana]  
Order of objects in **TreeSet** :[apple, banana, mango]  
Total time to insert 10M elements in **HashSet** in sec : **3564570637**  
Total time to insert 10M elements in **LinkedHashSet** in sec : **3511277551**  
Total time to insert 10M elements in **TreeSet** in sec : **10968043705**

**When to use HashSet, TreeSet and LinkedHashSet in Java**

Since all three implements Set interface they can be used for common Set operations like not allowing duplicates but since HashSet, TreeSet and LinkedHashSet has there special feature which makes them appropriate in certain scenario. Because of sorting order provided by TreeSet, use TreeSet when you need a collection where elements are sorted without duplicates. HashSet are rather general purpose Set implementation, Use it as default Set implementation if you need a fast, duplicate free collection. LinkedHashSet is extension of HashSet and its more suitable where you need to maintain **insertion order** of elements, similar to List without compromising performance for costly TreeSet. Another use of LinkedHashSet is for creating copies of existing Set, Since LinkedHashSet preservers insertion order, it returns Set which contains same elements in same order like exact copy. In short,  although all three are Set interface implementation they offer distinctive feature, HashSet is a general purpose Set while LinkedHashSet provides insertion order guarantee and TreeSet is a SortedSet which stores elements in sorted order specified by Comparator or Comparable in Java.

**How to copy object from one Set to other**

Here is code example of LinkedHashSet which demonstrate How LinkedHashSet can be used to copy objects from one Set to another without losing order. You will get exact replica of source Set, in terms of contents and order. Here static method copy(Set source) is written using Generics, This kind of parameterized method provides type-safety and help to avoid ClassCastException at runtime.

**import** java.util.Arrays;  
**import** java.util.HashSet;  
**import** java.util.LinkedHashSet;  
**import** java.util.Set;  
  
/\*\*  
 \* Java program to **copy object from one HashSet to another using LinkedHashSet**.  
 \* LinkedHashSet preserves order of element while copying elements.  
 \*   
 \* @author EmpId\_1011  
 \*/  
**public** **class** SetUtils{  
      
    **public** **static** **void** main(**String** args[]) {  
          
        **HashSet**<**String**> source = **new** **HashSet**<**String**>(**Arrays**.asList("Set, List, Map"));  
        **System**.out.println("source : " + source);  
        **Set**<**String**> copy = SetUtils.copy(source);  
        **System**.out.println("copy of HashSet using LinkedHashSet: " + copy);  
    }  
      
    */\*  
     \* Static utility method to copy Set in Java  
     \*/*  
    **public** **static** <T> **Set**<T> copy(**Set**<T> source){  
           **return** **new** **LinkedHashSet**<T>(source);  
    }  
}  
**Output:**  
source : [**Set**, **List**, **Map**]  
copy of **HashSet** using **LinkedHashSet**: [**Set**, **List**, **Map**]

Always code for interface than implementation so that you can replace HashSet to LinkedHashSet or TreeSet when your requirement changes.

Performance testing

The following method tests the performance of the three class on add() method.

|  |
| --- |
| **public** **static** **void** main(String[] args) {    Random r = **new** Random();    HashSet<Dog> hashSet = **new** HashSet<Dog>();  TreeSet<Dog> treeSet = **new** TreeSet<Dog>();  LinkedHashSet<Dog> linkedSet = **new** LinkedHashSet<Dog>();    *// start time*  **long** startTime = System.nanoTime();    **for** (**int** i = 0; i < 1000; i++) {  **int** x = r.nextInt(1000 - 10) + 10;  hashSet.add(**new** Dog(x));  }  *// end time*  **long** endTime = System.nanoTime();  **long** duration = endTime - startTime;  System.out.println("HashSet: " + duration);    *// start time*  startTime = System.nanoTime();  **for** (**int** i = 0; i < 1000; i++) {  **int** x = r.nextInt(1000 - 10) + 10;  treeSet.add(**new** Dog(x));  }  *// end time*  endTime = System.nanoTime();  duration = endTime - startTime;  System.out.println("TreeSet: " + duration);    *// start time*  startTime = System.nanoTime();  **for** (**int** i = 0; i < 1000; i++) {  **int** x = r.nextInt(1000 - 10) + 10;  linkedSet.add(**new** Dog(x));  }  *// end time*  endTime = System.nanoTime();  duration = endTime - startTime;  System.out.println("LinkedHashSet: " + duration);    } |

From the output below, we can clearly wee that HashSet is the fastest one.

HashSet: 2244768

TreeSet: 3549314

LinkedHashSet: 2263320

\* The test is not precise, but can reflect the basic idea that TreeSet is much slower because it is sorted.

