Modifying the Output Format for Set Classes in the Java Collections

WANWU GUO
School of Computer and Information Science
Edith Cowan University
2 Bradford Street, Mount Lawley, WA 6050
AUSTRALIA

Abstract – In Java Collections, the Set interface is implemented using a balanced BST and a chained HT as TreeSet class and HashSet class, respectively. TreeSet produces sorted output for the values in the set, but does not support the output for the values in their insertion order. HashSet provides neither sorted nor insertion-order output, but a random sequence of values. LinkedHashSet, a subclass of HashSet, supports the insertion-order outputs for the values in the set, but does not produce sorted output. This paper proposes an application-level solution and two class-level solutions for making these classes accommodate the sorted and insertion-order output together in their most efficient ways. The application-level solution requires no change to the current configurations of the set classes in the Java Collections, but its flexibility and portability are restricted to the level of application code only. The LinkedTreeSet solution requires a new class to be implemented using both a balanced BST and doubly linked list. The performance of its basic operations may be affected. By introducing an internal method for producing sorted output into the LinkedHashTable class, we can easily achieve the flexibility in output format without change any other configurations of the class. If this method is placed in the HashSet class, users can freely choose their preferred output format in random order, insertion order, or ascending order. This solution can also be applied to the hash map classes in the Java Collections.

Key-Words: - Data structures, Algorithm, Java Collections, Set classes, Output format, Flexibility, Portability

1 Introduction

Set is a collection of distinct values whose order is insignificant in constructing the collection. The concept of sets is widely used in many applications, such as a lottery pool of balls with distinct numbers, a group of students sitting randomly in a lecture theatre, and the membership of an association. Although the order of the values in a set is not significant in constructing the set, it is often useful to keep the output of a set in some predefined sequences. For example, the numbers drawn from a lottery pool may be in random order, but it is easier to check the drawn numbers against the entries in your ticket if the numbers are presented to you in ascending order. An association may have thousands of members. To list all the members, most likely the members are sorted using their names for easy search. Sometime it is more useful to organise the membership in the sequence of member’s joining order so that different membership grades can be easily allocated to individual members. This sequence is referred as the insertion order.

A set can be implemented using a number of data structures, such as arrays, linked lists, search tress, and hash tables. The two commonly used data structures for implementing sets are binary search trees (BSTs) and hash tables (HTs). In Java Collections, the Set interface is implemented using a balanced BST and a chained HT as TreeSet class and HashSet class, respectively. TreeSet produces sorted output for the values in the set, but does not support the output in insertion order. Provided by HashSet is neither sorted nor insertion-order output, but a random sequence of values. LinkedHashSet, a subclass of HashSet, supports the insertion-order output, but does not produce sorted output [1].

Based on the analysis of the three set classes included in the Java Collections, this paper proposes some solutions to these classes for making them to accommodate the sorted and insertion-order output together in their most efficient ways.

2 Set Classes in Java Collections

There are four set classes in the Java Collections, which are outlined in Figure 1 [1]. The AbstractSet class provides a skeletal implementation of the Set interface to minimize the effort required to implement this interface. It
extends, but does not override any of the implementations from the `AbstractCollection` class. It merely adds implementations for `equals()` and `hashCode()`.

The `TreeSet` class implements the `Set`, `Serializable`, `Cloneable`, `Iterable`, `Collection`, and `SortedSet` interfaces and extends the `AbstractSet` class using red-black trees, a kind of balanced BST. This balanced BST implementation provides guaranteed $O(\log n)$ time complexity for the basic operations such as `add()`, `remove()`, and `contains()`. This class guarantees that the sorted set will be in ascending order with a time complexity of $O(n)$.

```java
java.lang.Object
  ↓ java.util.AbstractCollection<E>
    ↓ java.util.AbstractSet<E>
      ↓ java.util.TreeSet<E>

a. The TreeSet class
```

The `HashSet` class implements the `Set`, `Serializable`, `Cloneable`, `Iterable`, and `Collection` interfaces and extends the `AbstractSet` class using chained hash tables. This class makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class offers $O(1)$ time performance for the basic operations such as `add()`, `remove()`, and `contains()`, assuming the hash function disperses the elements properly among the buckets. Iterating over this set is in the time complexity of $O(n+m)$ [1].

The `LinkedHashSet` class is a direct extension of the `HashSet` class. It uses a chained hash table and a doubly-linked list to implement the `Set` interface. This implementation differs from `HashSet` in that the doubly-linked list maintains all the elements in the order when they were inserted into the set, i.e., the insertion-order. Like `HashSet`, this class provides $O(1)$ performance for the basic operations, assuming the hash function disperses elements properly among the buckets. However, performance should be slightly slower than that of `HashSet`, due to the added expense of maintaining the doubly-linked list, except that iteration over a `LinkedHashSet` is only proportional to the size of the set, or $O(n)$, regardless of its capacity [1].

Figure 2 shows an example of the use of the three set classes to print out the set elements in their default order, respectively.

An empty set `InsertionSet` is created by `LinkedHashSet()` constructor at Line (1), and then six animals – Dog, Pig, Lion, Cat, Fox, and Cow – are inserted into this set in turn from Line (2) to Line (7). This set is then printed out at Line (9) in their insertion order.

A new empty sorted set `AscendingSet` is created using `TreeSet()` at Line (10). The method `addAll()` is called to copy all the elements in `InsertionSet` into `AscendingSet` at Line (11). This sorted set is then printed out at Line (13) in alphabetical order.

Another new empty set `RandomSet` is created using `HashSet()` at Line (14). The method `addAll()` is again called to copy all the elements in the sorted set `AscendingSet` into `RandomSet` at Line (15). Although the input is sorted, the elements in `RandomSet` are unsorted, which is proven at Line (17).

This example demonstrates that each of the three set classes in the Java Collections provides only one default output format. If the order of output plays no role in your use of the set classes in the Java Collections, the `HashSet` class is your best choice because it offers $O(1)$ performance for all the basic operations. Otherwise, if the sorted output is the only required format, the `TreeSet` class should be
preferred although its basic operations have a time complexity of $O(\log n)$. Remember sorting a set of $n$ values will have time complexity in the range of $O(n \log n)$ and $O(n^2)$ [2][3]. The `LinkedHashSet` class will be the only option if the output must be in insertion order.

It is evident that there is no class providing two or more options, from which the users can choose for their own needs on the output sequence of set elements.

```java
import java.util.*;

public class TestSets {
    public static void main (String[] args) {
        Set InsertionSet = new LinkedHashSet();
        InsertionSet.add("Dog");
        InsertionSet.add("Pig");
        InsertionSet.add("Lion");
        InsertionSet.add("Cat");
        InsertionSet.add("Fox");
        InsertionSet.add("Cow");
        System.out.println("Output in insertion order:");
        System.out.println("" + InsertionSet);
        SortedSet AscendingSet = new TreeSet();
        AscendingSet.addAll(InsertionSet);
        System.out.println("Output in ascending order:");
        System.out.println("" + AscendingSet);
        Set RandomSet = new HashSet();
        RandomSet.addAll(AscendingSet);
        System.out.println("Output with no particular order:");
        System.out.println("" + RandomSet);
    }
}
```

Output in insertion order:
[Dog, Pig, Lion, Cat, Fox, Cow]
Output in ascending order:
[Cat, Cow, Dog, Fox, Lion, Pig]
Output with no particular order:
[Lion, Cat, Pig, Fox, Cow, Dog]

Figure 2. An example demonstrating the output of set classes in Java Collections

### 3 Solutions for Output Flexibility in Using Set Classes in Java Collections

It is probably a reasonable assumption that the preferred output in most cases is in ascending order, and in some specific cases, the sequence of insertion-order is required. In very rare cases where random output is required, the `HashSet` should be sufficient to deal with it. Therefore, the flexible output for sets becomes enabling either `TreeSet` or `LinkedHashSet` to offer both sorted and insertion-order outcomes. This can be handled at either the application level or the class level.

At the application level, the programmer can use the methods available in both the `TreeSet` and `LinkedHashSet` to do conversion between a `LinkedHashSet` `InsertionSet` and a sorted set `AscendingSet`, being shown in Figure 2. The option of output format becomes a matter of calling either the `InsertionSet` or the `AscendingSet` for a specified purpose. This procedure can be formalised using the algorithm given in Figure 3.

It is critical to create the `LinkedHashSet` `InsertionSet` first to deal with the basic operations, such as insertion and deletion, because only such a set remembers the insertion sequence of elements. Step 4 creates a sorted set `AscendingSet` and then adds all the elements in `InsertionSet` to it. To that stage, two sets coexist in the application.

It should be noted that Step 4 is skipped if no sorted output is required. This not only saves memory resource, but also makes the application more efficient. Creating a BST at Step 4.2 takes
O(n\log n) time complexity, which is also the time complexity of this application-level algorithm.

To support set output in both sorted order and insertion order at application level:
1. Create a LinkedHashSet InsertionSet;
2. Add all elements into InsertionSet;
3. If insertion-order output is required,
   3.1 Print InsertionSet;
4. If ascending-order output is required,
   4.1 Create a TreeSet AscendingSet;
   4.2 Add all elements in InsertionSet into AscendingSet;
   4.3 Print AscendingSet;
5. Terminate.

Figure 3. The application level algorithm to support set output in both sorted order and insertion order

The application-level algorithm is not a generic portable solution for the programmers who would prefer to use the output options just like calling a method in a single set class. In other words, class-level solutions are likely to be more useful.

There are two options to make this happen: making the TreeSet class produce insertion-order output, and/or modifying the LinkedHashSet class to produce the ascending-order output.

To proceed with the first option, a LinkedTreeSet class must be implemented using the similar strategy adopted in the LinkedHashSet class. This will be an implementation by combining a balanced BST and a doubly linked list. When an element is added to such a set, it is inserted into its appropriate position in the BST, and also appended to the end of the doubly linked list. Two printing methods can be included into this class to offer choices between the ascending-order and insertion-order outcomes. The former will be linked to the BST only whereas the latter is solely associated with the list in the set.

This implementation has minimal effect on the performances of the major operations in the class, except that deletion may be in O(n) time complexity due to locating the specified element in the list before removing it. Both output methods have a time complexity of O(n). It should be noted that this LinkedTreeSet class uses more memory resource than the TreeSet class because it must maintain both a list and a BST of same set of elements to keep track of element insertion and deletion with the set.

The better option is to modify the LinkedHashSet class for supporting ascending-order output. This can be done easily by adding an extra method that prints the set elements in sorted order into the LinkedHashTable class. One solution to such a method is shown as an algorithm in Figure 4. We can copy all the set elements into an array, and then sort this array. The ascending order output simply becomes printing all the elements in the array.

To print the elements in LinkedHashSet LHSet in ascending order:
1. Create an array A1;
2. Copy all elements in LHSet into A1;
3. Sort array A1;
4. Print all elements in A1;
5. Terminate.

Figure 4. The algorithm to support ascending-order output in the LinkedHashSet class

An example of this class level solution is shown in Figure 5, which is a running of the program in Figure 1 by excluding the portion of the TreeSet usage and including a new method printAscendingOrder() that is implemented using the methods available in Java Arrays and HashSet classes. The sort method in Arrays implements a modified merge-sort algorithm, so it has a time complexity of O(n\log n) [1]. It should be noted that the array of set elements is not maintained all the time. It is active only once the printAscendingOrder() method is called.

4 Discussion and Conclusion

The application level solution to flexible output format requires no change to the current configurations of the set classes in the Java Collections, but its flexibility and portability are restricted to the level of application code.

The LinkedTreeSet solution requires a new class to be implemented using both a balanced BST and doubly linked list. This class needs to maintain the two copies of the elements separately in the BST and list. The performance of its basic operations, especially deletion, may be affected negatively.

By introducing an internal method for producing sorted output into the LinkedHashTable class, we can easily achieve the flexibility in output format without change any other configurations of the class. Although this method seems slower in returning the sorted outcome, it does not affect the performance of the basic operations in that class. Remember that those basic operations are frequently called by the applications. The outcome is likely required only after all the elements are settled down in the
application, so it is likely to be a once-per-run operation. Therefore, its performance of $O(n \log n)$ is acceptable.

If this method is placed in the HashSet class, both the HashSet and LinkedHashSet class are able to produce sorted outcome with Java’s inheritance property. Thus, with this inclusion, users can freely choose their preferred output format in random order, insertion order, or ascending order. This solution can also be applied to the hash map classes in the Java Collections.

```java
import java.util.*;

public class TestHashSets {
    public static void main (String[] args) {
        Set LHS = new LinkedHashSet();
        LHS.add("Dog");
        LHS.add("Pig");
        LHS.add("Lion");
        LHS.add("Cat");
        LHS.add("Fox");
        LHS.add("Cow");
        System.out.println("Set elements in insertion order:");
        System.out.println("" + LHS);
        System.out.println();
        printAscendingOrder(LHS);
        Set HS = new HashSet();
        HS.addAll(LHS);
        System.out.println("Set elements in random order:");
        System.out.println("" + HS);
        System.out.println();
        printAscendingOrder(HS);
        System.out.println();
    }

    private static void printAscendingOrder( Set S) {
        Object[] A1 = S.toArray();
        Arrays.sort(A1);
        System.out.println("Set elements in ascending order:");
        System.out.print("[");
        for (int i = 0; i < A1.length-1; i++)
            System.out.print(A1[i] + ", ");
        System.out.println(A1[A1.length-1]);
        System.out.println();
    }
}
```

Set elements in insertion order:
[Dog, Pig, Lion, Cat, Fox, Cow]
Set elements in ascending order:
[Cat, Cow, Dog, Fox, Lion, Pig]
Set elements in random order:
[Cat, Lion, Pig, Fox, Cow, Dog]
Set elements in ascending order:
[Cat, Cow, Dog, Fox, Lion, Pig]

Figure 5. An example demonstrating the flexible output of the hash set classes in Java Collections

References: