3/4/2023

Java Collections

Linked HashMap, Tree HashMap, Comparable and Comparator Interface

Submitted by:

Finla N A - 245242

Sagar Saji - 245195

Kevin J Kodiyan - 245118

Migha Maria Joseph - 245137

LinkedHashMap

Introduction to LinkedHashmap

A LinkedHashMap is a subclass of the HashMap class in Java. It maintains a doubly linked list of the entries in the map, allowing for iteration in the order in which the entries were added. This is in contrast to a regular HashMap, which does not guarantee any specific order when iterating over its entries. LinkedHashMap is particularly useful when you need to maintain the order in which entries were added to the map, as well as when you need constant time O(1) access to entries in the map.

In this documentation, we will explore the various aspects of the LinkedHashMap class, including its constructors, methods, and some examples of how to use it.

Constructors:

The LinkedHashMap class provides four constructors for creating instances of the class. Here is a brief overview of each of them:

- LinkedHashMap(): This constructor creates an empty LinkedHashMap with the default initial capacity (16) and the default load factor (0.75).
- LinkedHashMap(int initialCapacity): This constructor creates an empty LinkedHashMap with the specified initial capacity and the default load factor (0.75).
- LinkedHashMap(int initialCapacity, float loadFactor): This constructor creates an empty LinkedHashMap with the specified initial capacity and load factor.
- LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder): This constructor creates an empty LinkedHashMap with the specified initial capacity, load factor, and access order.

Methods:

The LinkedHashMap class provides many methods for manipulating and accessing the entries in the map. Here are some of the most commonly used methods:

- put(K key, V value): This method adds a new key-value pair to the map. If the key already exists in the map, the value associated with that key is replaced with the new value.
- get(Object key): This method returns the value associated with the specified key in the map. If the key is not present in the map, null is returned.
- remove(Object key): This method removes the entry associated with the specified key from the map. If the key is not present in the map, null is returned.
- clear(): This method removes all entries from the map.
- containsKey(Object key): This method returns true if the specified key is present in the map, false otherwise.
- containsValue(Object value): This method returns true if the specified value is present in the map, false otherwise.
- size(): This method returns the number of entries in the map.
- keySet(): This method returns a Set containing all the keys in the map.
- values(): This method returns a Collection containing all the values in the map.
- entrySet(): This method returns a Set containing all the entries (key-value pairs) in the map.

Access Order vs Insertion Order:

As mentioned earlier, a LinkedHashMap maintains a doubly linked list of the entries in the map. By default, this list is maintained in the order in which the entries were inserted into the map. This is known as insertion order.

However, the LinkedHashMap class also provides the option of maintaining the list in access order. In access order, the most recently accessed entry is moved to the end of the list. This is particularly useful in scenarios where you need to implement a cache, where you want to remove the least recently accessed item when the cache becomes full.

To enable access order, you can use the following constructor:

LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)

Setting accessOrder to true enables access order, while setting it to false (the default) maintains insertion order.

Example 1: Insertion Order

Let's look at a simple example of using a LinkedHashMap to maintain insertion order

```
import java.util.LinkedHashMap;

import java.util.Map

public class LinkedHashMapExample

public static void main(String[] args)

 {

        Map<String, Integer> map = new LinkedHashMap<>();

        // Add some entries to the map

        map.put("one", 1);

        map.put("two", 2);

        map.put("three", 3);

        map.put("four", 4);

        for (Map.Entry<String, Integer> entry : map.entrySet())// Iterate over the entries in the map

        {

                System.out.println(entry.getKey() + ": " + entry.getValue());

        }}
```

Output:

one: 1

two: 2

three: 3

four: 4

In this example, we create a new LinkedHashMap and add some entries to it. When we iterate over the entries in the map using the entrySet() method, the entries are returned in the order in which they were added to the map.

Example 2: Access Order

Now let's look at an example of using a LinkedHashMap to maintain access order.

```
import java.util.LinkedHashMap;

import java.util.Map;

public class LinkedHashMapExample

{

public static void main(String[] args)

{

        // Create a new LinkedHashMap with access order

        Map<String, Integer> map = new LinkedHashMap<>(16, 0.75f, true);

        // Add some entries to the map

        map.put("one", 1);
```

```
        map.put("two", 2);
        map.put("three", 3);
        map.put("four", 4);

        // Access some entries to change their order
        map.get("one");
        map.get("two");

        // Iterate over the entries in the map
        for (Map.Entry<String, Integer> entry : map.entrySet()) {
        System.out.println(entry.getKey() + ": " + entry.getValue());
        }
}
}
```

Output:

```
three: 3
four: 4
one: 1
two: 2
```

In this example, we create a new LinkedHashMap with access order by setting the accessOrder parameter to true in the constructor. We add some entries to the map and then access some of them using the get() method. This changes the order of the entries in the map, with the most recently accessed entries moved to the end of the list.

When we iterate over the entries in the map using the entrySet() method, they are returned in access order, with the most recently accessed entries returned last.

Additional features of the LinkedHashMap :

- Removing Eldest Entries: The LinkedHashMap class also provides a protected method called removeEldestEntry() that can be overridden to provide a custom policy for removing the eldest (i.e., oldest or least recently accessed) entry in the map. This can be useful when implementing a cache or LRU (least recently used) algorithm.

- Serialization: The LinkedHashMap class also implements the Serializable interface, which means that it can be serialized and deserialized using Java's built-in serialization mechanism. This can be useful when storing maps to disk or transmitting them over a network.

- Performance: It is worth noting that the LinkedHashMap class has slightly worse performance than the regular HashMap class due to the extra overhead of maintaining the linked list. However, this performance difference is usually negligible and only becomes significant when working with very large maps.

Some practices when using the Linked HashMap :

- Use the LinkedHashMap class when you need to maintain the order of entries in a map. If you don't need to maintain the order, it's usually better to use a regular HashMap or other map implementation that is optimized for speed and memory usage.

- Choose the order you need carefully. If you need to maintain insertion order, use the default constructor. If you need to maintain access order, use the constructor that takes an accessOrder parameter. Using the wrong order can lead to unexpected behavior and performance problems.

- Be aware of the performance implications of using a LinkedHashMap. While the performance difference between a LinkedHashMap and a regular HashMap is usually small, it can become significant when working with very large maps. If performance is a concern, be sure to benchmark your code and consider other map implementations like TreeMap or ConcurrentHashMap. By following these best practices, you can ensure that your code is efficient, maintainable, and bug-free when using the LinkedHashMap class in Java

By following these best practices, you can ensure that your code is efficient, maintainable, and bug-free when using the LinkedHashMap class in Java

Conclusion:

In this documentation, we have explored the various aspects of the Linked HashMap class in Java. We have seen how it can be used to maintain the order of entries in a map, We have also seen some examples of how to use the LinkedHashMap class in practice.

Overall, the LinkedHashMap class is a useful tool for maintaining order in maps in Java, and is worth considering when designing applications that require ordered collections.

In conclusion, the LinkedHashMap class in Java provides a powerful and flexible way to maintain the order of entries in a map. It is easy to use and provides a variety of options for controlling the order, including insertion order and access order.

Tree HashMap

Introduction

Java provides several data structures for storing and managing key-value pairs, including HashMap, TreeMap, and LinkedHashMap. However, these data structures have their own strengths and weaknesses. For example, HashMap provides fast lookup times, but the order of the elements is not guaranteed. TreeMap provides ordered iteration, but the lookup time is slower. To combine the benefits of both data structures, Java introduced the TreeHashMap class, which is a combination of TreeMap and HashMap.

Tree HashMap class:

The TreeHashMap class is a part of the Java Collections Framework and is defined in the java.util package. It extends the AbstractMap class and implements the Map interface. The TreeHashMap class is similar to the HashMap and TreeMap classes, but it combines the features of both data structures to provide the best of both worlds.

The TreeHashMap class uses a HashMap to store the key-value pairs and a TreeMap to maintain the order of the elements. The keys are stored in the HashMap, and the TreeMap is used to maintain the order of the keys. This allows for fast lookup times and ordered iteration.

Declaration:

To declare a new TreeHashMap, you can use the following syntax:

```
TreeMap<KeyType, ValueType> treeHashMap = new TreeMap<>();
```

In this syntax, KeyType is the data type of the keys, and ValueType is the data type of the values. For example, if you want to store String keys and Integer values, you can declare a TreeHashMap as follows:

```
TreeMap<String, Integer> treeHashMap = new TreeMap<>();
```

Adding and updating elements:

To add elements to a TreeHashMap, you can use the put() method, which takes a key and a value as arguments and adds the key-value pair to the TreeHashMap. If the key already exists in the TreeHashMap, the value associated with the key is updated.

Here is an example of how to add and update elements in a TreeHashMap:

```
import java.util.TreeMap;

public class Example {
    public static void main(String[] args) {
        TreeMap<String, Integer> treeHashMap = new TreeMap<>();

        treeHashMap.put("apple", 1);
        treeHashMap.put("orange", 2);
        treeHashMap.put("banana", 3);

        Integer previousValue = treeHashMap.put("apple", 4);
        System.out.println(previousValue); // Output: 1
        System.out.println(treeHashMap.get("apple")); // Output: 4
    }
}
```

In this example, we first declare a new TreeHashMap with String keys and Integer values. We add three key-value pairs to the TreeHashMap using the put() method. We then add a new key-value pair with the key "apple" and the value 4 using the put() method. The put() method returns the previous value associated with the key "apple", which is 1. We print the previous value using System.out.println(). Finally, we retrieve the

value associated with the key "apple" using the get() method, which returns 4.

Retrieving elements:

To retrieve elements from a TreeHashMap, you can use the get() method, which takes a key as an argument and returns the value associated with the key, or null if there is no value associated with the key.

Here is an example of how to retrieve elements from a TreeHashMap:

```
import java.util.TreeMap;

public class Example {
    public static void main(String[] args) {
        TreeMap<String, Integer> treeHashMap = new TreeMap<>();

        treeHashMap.put("apple", 1);
        treeHashMap.put("orange", 2);
        treeHashMap.put("banana", 3);

        Integer value = treeHashMap.get("orange");
        System.out.println(value); // Output:
```

In this example, we first declare a new TreeHashMap with String keys and Integer values. We add three key-value pairs to the TreeHashMap using the put() method. We then retrieve the value associated with the key "orange" using the get() method, which returns 2. We print the value using System.out.println().

Iterating over elements:

To iterate over the elements in a TreeHashMap, you can use the entrySet() method, which returns a set of key-value pairs. You can then use a for-each loop to iterate over the set and retrieve the keys and values.

Here is an example of how to iterate over the elements in a TreeHashMap

```
import java.util.Map;
import java.util.TreeMap;

public class Example {
    public static void main(String[] args) {
        TreeMap<String, Integer> treeHashMap = new TreeMap<>();

        treeHashMap.put("apple", 1);
        treeHashMap.put("orange", 2);
        treeHashMap.put("banana", 3);

        for (Map.Entry<String, Integer> entry : treeHashMap.entrySet()) {
            System.out.println(entry.getKey() + ": " + entry.getValue());
        }
    }
}
```

In this example, we first declare a new TreeHashMap with String keys and Integer values. We add three key-value pairs to the TreeHashMap using the put() method. We then retrieve the value associated with the

key "orange" using the get() method, which returns 2. We print the value using System.out.println().

Iterating over elements:

To iterate over the elements in a TreeHashMap, you can use the entrySet() method, which returns a set of key-value pairs. You can then use a for-each loop to iterate over the set and retrieve the keys and values.

Here is an example of how to iterate over the elements in a TreeHashMap:

```
import java.util.Map;
import java.util.TreeMap;

public class Example {
    public static void main(String[] args) {
        TreeMap<String, Integer> treeHashMap = new TreeMap<>();

        treeHashMap.put("apple", 1);
        treeHashMap.put("orange", 2);
        treeHashMap.put("banana", 3);

        for (Map.Entry<String, Integer> entry : treeHashMap.entrySet()) {
            System.out.println(entry.getKey() + ": " + entry.getValue());
        }
    }
}
```

In this example, we first declare a new TreeHashMap with String keys and Integer values. We add three key-value pairs to the TreeHashMap using the put() method. We then retrieve the set of key-value pairs using the entrySet() method and iterate over the set using a for-each loop. For each element in the set, we retrieve the key and value using the getKey() and getValue() methods of the Map.Entry interface, and we print them using System.out.println().

Removing elements:

To remove elements from a TreeHashMap, you can use the remove() method, which takes a key as an argument and removes the key-value pair associated with the key.

Here is an example of how to remove elements from a TreeHashMap

```
import java.util.TreeMap;

public class Example {
    public static void main(String[] args) {
        TreeMap<String, Integer> treeHashMap = new TreeMap<>();

        treeHashMap.put("apple", 1);
        treeHashMap.put("orange", 2);
        treeHashMap.put("banana", 3);

        treeHashMap.remove("orange");

        System.out.println(treeHashMap.containsKey("orange")); // Output: false
    }
}
```

In this example, we first declare a new TreeHashMap with String keys and Integer values. We add three key-value pairs to the TreeHashMap using the put() method. We then remove the key-value pair associated with the key "orange" using the remove() method. We check if the key "orange" is still present in the TreeHashMap using the containsKey() method, which returns false.

Sorting elements:

One of the main advantages of the TreeHashMap class is that it maintains the order of the elements. The elements in a TreeHashMap are sorted based on the natural ordering of the keys, or based on a Comparator object that is provided when the TreeHashMap is created.

Here is an example of how to sort the elements in a TreeHashMap using a Comparator

```
import java.util.Comparator;
import java.util.TreeMap;

public class Example {
    public static void main(String[] args) {
        TreeMap<String, Integer> treeHashMap = new
TreeMap<>(Comparator.reverseOrder());

        treeHashMap.put("apple", 1);
        treeHashMap.put("orange", 2);
        treeHashMap.put("banana", 3);

        for (String key : treeHashMap.keySet()) {
            System.out.println(key + ": " + treeHashMap.get(key));
        }
    }
}
```

In this example, we first declare a new TreeHashMap with String keys and Integer values, and we provide a Comparator that sorts the elements in reverse order. We add three key-value pairs to the TreeHashMap using the put() method. We then retrieve the keys using the keySet() method and iterate over them using a for-each loop. For each key, we retrieve the corresponding value using the get() method, and we print them using System.out.println().

Performance considerations:

The performance of a TreeHashMap depends on the size of the map, the distribution of the keys, and the implementation of the Comparator if one is provided. In general, the TreeHashMap class provides O(log n) performance for most operations, including insertion, deletion, and retrieval. However, in some cases, the performance can be worse, especially if the keys are not well-distributed.

If you are working with a large TreeHashMap, you should be careful about the memory usage. Each element in the map takes up some memory, and if you add too many elements, you can quickly run out of memory. You can monitor the memory usage using a profiler or by printing the memory usage before and after adding elements.

Another performance consideration is the implementation of the Comparator. If you provide a Comparator that is expensive to compute or that does not distribute the keys well, the performance of the TreeHashMap can be significantly worse. Therefore, you should choose a Comparator that is efficient and that distributes the keys well.

Conclusion:

The TreeHashMap class in Java is a powerful data structure that provides a sorted map of key-value pairs. It is based on the TreeMap class and provides similar functionality, but with some differences in terms of syntax and performance. The TreeHashMap is useful when you need to maintain the order of the elements or when you need to search for elements based on their keys. It is also useful when you need to perform range searches, as it provides efficient methods for finding the elements within a range of keys. However, you should be careful about the performance and memory usage when working with large TreeHashMaps, and you should choose a Comparator that is efficient and that distributes the keys well.

Comparable Interface

Introduction:

A comparable interface in Java is an interface that allows us to sort an array or a collection of objects based on a certain criterion. It defines a single method called "compareTo" which is used to compare two objects. The "compareTo" method returns an integer value that indicates the relationship between the two objects being compared.

Comparable Interface

The Comparable interface is defined in the java.lang package and contains only one method called "compareTo". This method compares the current object with another object and returns an integer value based on the comparison result. The compareTo method takes a single argument, which is the object to be compared with the current object. Syntax: The syntax of the compareTo method is as follows:

```
public int compareTo(Object o)
```

The return value of the compareTo method is an integer value that indicates the relationship between the two objects being compared. The possible return values are:

- A negative integer: If the current object is less than the object being compared to.
- Zero: If the current object is equal to the object being compared to.
- A positive integer: If the current object is greater than the object being compared to.

The compareTo method throws a ClassCastException if the object being compared is not of the same type as the current object. Example: Let's consider an example where we want to sort an array

of objects based on the age of the person. We will create a Person class that implements the Comparable interface and provides an implementation for the compareTo method. The implementation of the compareTo method compares the age of two Person objects and returns a negative, zero, or positive integer based on the comparison result.

```
public class Person implements Comparable<Person> {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    @Override
    public int compareTo(Person person) {
        return this.age - person.age;
    }
}
```

In the above example, we have created a Person class that implements the Comparable interface. The implementation of the compareTo method compares the age of two Person objects and

returns a negative, zero, or positive integer based on the comparison result.

Sorting Objects

Once we have implemented the Comparable interface for our class, we can use the Arrays.sort() method or the Collections.sort() method to sort an array or a collection of objects based on the compareTo method. The Arrays.sort() method and the Collections.sort() method both use the natural ordering of objects, which is based on the implementation of the compareTo method. Let's consider an example where we want to sort an array of Person objects based on the age of the person. We will create an array of Person objects and use the Arrays.sort() method to sort the array based on the age of the person.

```
public class Main {
    public static void main(String[] args) {
        Person[] persons = {
                new Person("John", 30),
                new Person("David", 25),
                new Person("Sarah", 35)
        };
        Arrays.sort(persons);
        for (Person person : persons) {
            System.out.println(person.getName() + " " + person.getAge());
        }
    }
}
```

In the above example, we have created an array of Person objects and used the Arrays.sort() method to sort the array based on the age of the person. The output of the above code will be:

```
David 25
John 30
Sarah 35
```

Comparator Interface

Introduction

The Comparator interface in Java is an interface that allows us to define a custom order for objects based on certain criteria. Unlike the Comparable interface, which defines a natural ordering for objects, the Comparator interface allows us to sort objects based on a specific criterion that we define.

Comparator Interface

The Comparator interface is defined in the java.util package and contains two methods: "compare" and "equals". The "compare" method compares two objects and returns an integer value that indicates the relationship between the two objects being compared. The "equals" method checks whether two Comparator objects are equal.

Syntax

The syntax of the "compare" method is as follows:

```
public int compare(Object o1, Object o2)
```

The "compare" method takes two arguments, which are the objects to be compared. The return value of the "compare" method is an integer value that indicates the relationship between the two objects being compared. The possible return values are: A negative integer: If the first object is less than the second object. Zero: If the first object is equal to the second object. A positive integer: If the first object is greater than the second object. The "compare" method throws a ClassCastException if the objects being compared are not of the same type. The "equals" method checks whether two Comparator objects are equal.

The syntax of the "equals" method is as follows:

```
public boolean equals(Object obj)
```

The "equals" method takes a single argument, which is the object to be compared with the current object. The "equals" method returns true if the two Comparator objects are equal and false otherwise.

Example: Let's consider an example where we want to sort an array of objects based on the name of the person. We will create a Person class and a NameComparator class that implements the Comparator interface and provides an implementation for the compare method. The implementation of the compare method compares the name of two Person objects and returns a negative, zero, or positive integer based on the comparison result.

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}

public class NameComparator implements Comparator<Person> {
    @Override
```

```
    public int compare(Person p1, Person p2) {
        return p1.getName().compareTo(p2.getName());
    }
}
```

In the above example, we have created a Person class and a NameComparator class that implements the Comparator interface. The implementation of the compare method compares the name of two Person objects and returns a negative, zero, or positive integer based on the comparison result.

Sorting Objects

Once we have implemented the Comparator interface for our class, we can use the Arrays.sort() method or the Collections.sort() method to sort an array or a collection of objects based on the compare method. The Arrays.sort() method and the Collections.sort() method both take a second argument, which is an instance of the Comparator class.

Let's consider an example where we want to sort an array of strings based on the length of the string. We will create an array of strings and use the Arrays.sort() method to sort the array based on the length of the string using the StringLengthComparator.

```
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        String[] strings = {"apple", "orange", "banana", "kiwi"};

        Arrays.sort(strings, new StringLengthComparator());

        for (String string : strings) {
            System.out.println(string);
```

```
    }
  }
}
```

In the above example, we have created an array of strings and used the Arrays.sort() method to sort the array based on the length of the string using the StringLengthComparator. The output of the above code will be:

```
kiwi
apple
banana
orange
```

Conclusion

So here we have discussed about Comparator interfaces and how to use this interface over different classes. We can implement on multiple classes and these classes could be used at the same program for comparison and sorting based on different data members. Also we could use Collections.sort() inorder to sort the elements from a list or class