Object - Oriented Programming 2

Week 12. TreeSet Class, Map Interface, HashMap Class, TreeMap Class

By Elubu Joseph - MSc.IS

Lecturer

Department of Information Technology

Kumi University

Email: josebulinda@gmail.com

jose@kumiuniversity.ac.ug

Agenda

1. TreeSet Class,
2. Map Interface,
3. HashMap Class,
4. TreeMap Class

TreeSet Class

TreeSet Class

Is used to store unique elements in ascending order. It is similar to **HashSet** except that it sorts the elements in the ascending order while HashSet doesn't maintain any order.

TreeSet class implements the Set interface and use tree based data structure storage. It extends AbstractSet class and implements the NavigableSet interface. The declaration of the class is given below.

Declaration

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

TreeSet Class + Features

1.stores the elements in ascending order.

2.uses a Tree structure to store elements.

3.contains unique elements only, like HashSet.

4.access and retrieval times are quite fast.

5.doesn't allow null element.

6.is non synchronized.

TreeSet Class+Constructors

TreeSet class has 4 Constructors

```
TreeSet()
TreeSet( Collection C )
TreeSet( Comparator comp )
TreeSet( SortedSet ss )
```

TreeSet Methods

Method	Description
boolean **add**(E e)	adds the specified element to this set if it is not already present.
boolean **addAll**(Collection<? extends E> c)	adds all of the elements in the specified collection to this set.
E **ceiling**(E e)	returns the equal or closest greatest element of the specified element from the set, or null if there is no such element.
Comparator<? super E> **comparator**()	returns comparator that arranged elements in order.
Iterator **descendingIterator**()	is used iterate the elements in descending order.
NavigableSet **descendingSet**()	returns the elements in reverse order.
E **floor**(E e)	returns the equal or closest least element of the specified element from the set, or null if there is no such element.
SortedSet **headSet**(E toElement)	returns the group of elements that are less than the specified element.

TreeSet Methods+

Method	Description
E **higher**(E e)	returns the closest greatest element of the specified element from the set, or null there is no such element.
Iterator **iterator**()	is used to iterate the elements in ascending order.
E **lower**(E e)	returns the closest least element of the specified element from the set, or null there is no such element.
E **pollFirst**()	is used to retrieve and remove the lowest(first) element.
E **pollLast**()	is used to retrieve and remove the highest(last) element.
Spliterator **spliterator**()	is used to create a late-binding and fail-fast spliterator over the elements.
NavigableSet **subSet**(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)	returns a set of elements that lie between the given range.

TreeSet Methods++

Method	Description
boolean **contains**(Object o)	returns true if this set contains the specified element.
boolean **isEmpty**()	returns true if this set contains no elements.
boolean **remove**(Object o)	is used to remove the specified element from this set if it is present.
void **clear**()	is used to remove all of the elements from this set.
Object **clone**()	returns a shallow copy of this TreeSet instance.
E **first**()	returns the first (lowest) element currently in this sorted set.
E **last**()	returns the last (highest) element currently in this sorted set.
int **size**()	returns the number of elements in this set.

Example Program-to add Elements To TreeSet

Lets take an example to create a TreeSet that contains duplicate elements. But you can notice that it prints unique elements that means it does not allow duplicate elements.

```
import java.util.*;
class TSet1 {
	public static void main(String[] args) {
	TreeSet< String> al = new TreeSet< String>();
	al.add("BIT"); al.add("MBA");
	al.add("MBA"); al.add("MIS");
	al.add("BTHEO");
	System.out.println("Inserted Elements\n"+ al);
	Iterator it=al.iterator();

	while(it.hasNext()){

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

	}

} }
```

Example Program-to add Elements To TreeSet+code and output

```
1   package CollectionClass;
2   import java.util.*;
3   class TSet {
4           public static void main(String[] args) {
            TreeSet< String> al = new TreeSet< String>();
6           al.add("BIT"); al.add("MBA");
7           al.add("MBA"); al.add("MIS");
8           al.add("BTHEO");
9           System.out.println("Inserted Elements\n"+ al);
10          Iterator it=al.iterator();
11
12          while(it.hasNext()){
13              System.out.println(it.next());
14          }
15  } }
```

```
run:
Inserted Elements
[BIT, BTHEO, MBA, MIS]
BIT
BTHEO
MBA
MIS
```

Removing Elements From the TreeSet

We can use remove() method of this class to remove the elements. Note you will have to remove elements by their names not location number since this remove() method takes in string. See the below example.

```
import java.util.*;
class TSetR_Element {
        public static void main(String[] args) {
        TreeSet< String> al = new TreeSet< String>();
        al.add("BIT"); al.add("MBA");
        al.add("MBA"); al.add("MIS");
        al.add("BTHEO");
        System.out.println("Inserted Elements\n"+ al);
        Iterator it=al.iterator();

        while(it.hasNext()){
        System.out.println(it.next());

        }
        al.remove("BTHEO");
        System.out.println("After removing Elements\n"+ al);
} }
```

Removing Elements From the TreeSet+code and output

```
1   package CollectionClass;
2   import java.util.*;
3   class TSetR_Element {
4           public static void main(String[] args) {
5           TreeSet< String> al = new TreeSet< String>();
6           al.add("BIT"); al.add("MBA");
7           al.add("MBA"); al.add("MIS");
8           al.add("BTHEO");
9           System.out.println("Inserted Elements\n"+ al);
10          Iterator it=al.iterator();
11
12          while(it.hasNext()){
13          System.out.println(it.next());
14          }
15           al.remove("BTHEO");
16           System.out.println("After removing 1 Element\n"+ al);
17  } }
```

```
run:
Inserted Elements
[BIT, BTHEO, MBA, MIS]
BIT
BTHEO
MBA
MIS
After removing 1 Element
[BIT, MBA, MIS]
```

Search an Element in TreeSet

TreeSet provides contains() method that returns true if elements is present in the set.

To search the element we have to include the following code.

```
boolean searchE = al.contains(“Name of element here");
System.out.println("Is contain ?: "+ searchE);
```

Traverse TreeSet in Ascending and Descending Order

We can traverse elements of Treeset in both ascending and descending order. TreeSet provides descendingIterator() m ethod that returns iterator type for descending traversing. See the below example.

```
System.out.println("Ascending order");
Iterator it=al.iterator();
      while(it.hasNext()){
      System.out.println(it.next());
      }
System.out.println("Descending order");
Iterator it1=al.descendingIterator();
      while(it1.hasNext()){
      System.out.println(it1.next());
      }
```

Complete Example of Search and Traversing TreeSet in Assending and descending order

```
import java.util.*;
class STTSet{
        public static void main(String[] args) {
        TreeSet< String> al = new TreeSet< String>();
        al.add("BIT"); al.add("MBA");
        al.add("MBA"); al.add("MIS");
        al.add("BTHEO");
        System.out.println("Ascending order");
        Iterator it=al.iterator();
        while(it.hasNext()){
        System.out.println(it.next());
        }
        boolean searchE = al.contains("MIS");
        System.out.println("The Set contain MIS: "+searchE);
        System.out.println("Descending order");
        Iterator it1=al.descendingIterator();
        while(it1.hasNext()){
        System.out.println(it1.next());
        }
} }
```

By default Iterator processes elements in Ascending order.

Searching for an Element called MIS from the set.

Iteratorating elements using descendingIterator(); processes elements in descending order.

Complete Example of Search and Traversing TreeSet in Assending and descending order + code and output

```
1   package CollectionClass;
2   import java.util.*;
3   class STTSet{
4           public static void main(String[] args) {
            TreeSet< String> al = new TreeSet< String>();
6           al.add("BIT"); al.add("MBA");
7           al.add("MBA"); al.add("MIS");
8           al.add("BTHEO");
9           System.out.println("Ascending order");
10          Iterator it=al.iterator();
11          while(it.hasNext()){
12          System.out.println(it.next());
13          }
14          boolean searchE = al.contains("MIS");
15          System.out.println("The Set contain MIS: "+searchE);
16          System.out.println("Descending order");
17          Iterator it1=al.descendingIterator();
18          while(it1.hasNext()){
19          System.out.println(it1.next());
20          }
21  } }
```

```
run:
Ascending order
BIT
BTHEO
MBA
MIS
The Set contain MIS: true
Descending order
MIS
MBA
BTHEO
BIT
```

Map Interface

Map Interface

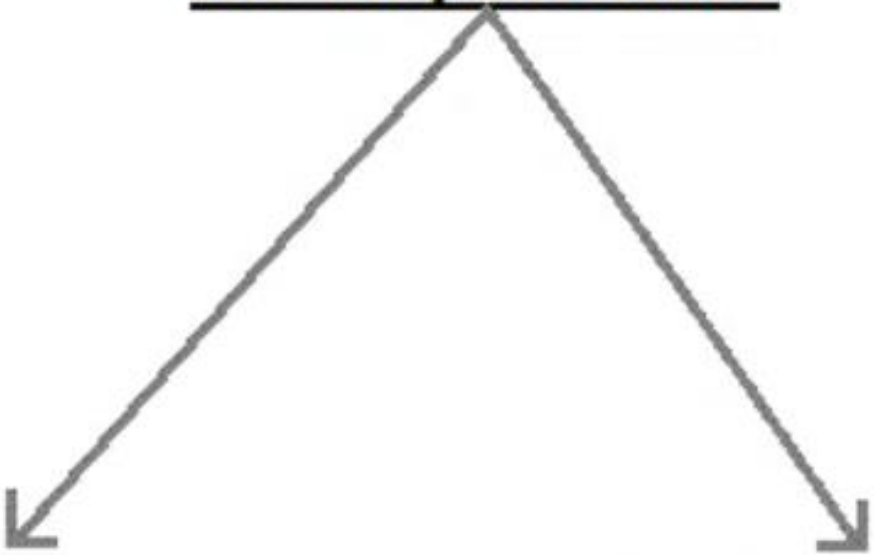
An interface that stores data in key and value association where both key and values are objects. The **key must be unique** but the **values can be duplicate**. Although Maps are a part of Collection Framework, they can not actually be called as collections because of some properties that they posses.

However we can obtain a **collection-view** of maps. It provides various classes: **HashMap, TreeMap, LinkedHashMap** for map implementation. All these classes implements Map interface to provide Map properties to the collection.

Map Interface and its Sub-interface

Interface	Description
Map	Maps unique key to value.
Map.Entry	Describe an element in key and value pair in a map. Entry is sub interface of Map.
NavigableMap	Extends SortedMap to handle the retrival of entries based on closest match searches
SortedMap	Extends Map so that key are maintained in an ascending order.

The Map Interface



SortedMap Interface

- Extends Map interface.
- Ensures that the entries are in ascending order based on the keys.

NavigableMap Interface

- Extends SortedMap interface.
- Declares behaviour that supports retrieval of entries based on closest match.

(Studytonight.com)

Map Interface Methods

These are commonly used methods defined by Map interface

1. boolean **containsKey**(Object *k*): returns true if map contain *k* as key. Otherwise false.
2. Object **get**(Object *k*) : returns values associated with the key *k*.
3. Object **put**(Object *k*, Object *v*) : stores an entry in map.
4. Object **putAll**(Map *m*) : put all entries from *m* in this map.
5. Set **keySet**() : returns **Set** that contains the key in a map.
6. Set **entrySet**() : returns **Set** that contains the entries in a map.

HashMap Class

HashMap Class

Java HashMap class is an implementation of Map interface based on hash table. It stores elements in key & value pairs which is denoted as HashMap<Key, Value> or HashMap<K, V>.

It extends AbstractMap class, and implements Map interface and can be accessed by importing **java.util** package. Declaration of this class is given below.

Declaration

```
public class HashMap<K,V>extends AbstractMap<K,V>implements
Map<K,V>,Cloneable, Serializable
```

HashMap Class+

Important features:

1. HashMap class is member of the Java Collection Framework.
2. It uses a hashtable to store the map. This allows the execution time of get() and put() to remain same.
3. HashMap does not maintain order of its element.
4. It contains values based on the key.
5. It allows only unique keys.
6. It is unsynchronized.
7. Its initial default capacity is 16.
8. It permits null values and the null key

HashMap Class Constructors

provides following 4 constructors.

```
HashMap()
HashMap(Map< ? extends k, ? extends V> m)
HashMap(int capacity)
HashMap(int capacity, float loadfactor)
```

Example: Creating and Adding Elements To HashMap

In this example, we will create a HashMap that can store integer type key and string values. HashMap provides put() method that takes two arguments first is key and second is value. See the below example.

Example: Creating and Adding Elements to HashMap+code and output

```
import java.util.*;

class HMPro {
public static void main(String args[]){
      HashMap< Integer,String> hm = new HashMap< Integer,String >();
      hm.put(1, "A"); hm.put(200, " K");
      hm.put(3,"C"); hm.put(400, "Y");
      // Displaying HashMap
      System.out.println(hm);
}}
```

```
run:
{400=Y, 1=A, 3=C, 200= K}
```

Traversing and Removing Elements From HashMap

To access elements of the HashMap, we can traverse them using the loop. In this example, we are using for loop to iterate the elements.

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

In case, we need to remove any element from the HashMap. We can use remove() method that takes key as an argument. It has one overloaded remove() method that takes two arguments first is key and second is value.

```
hm.remove(2); System.out.println("After Removing 2 :\n"+hm);
```

Traversing and Removing Elements From HashMap**+Code**

```
import java.util.*;

class TraverseHM {
public static void main(String args[]){
        HashMap< Integer,String> hm = new HashMap< Integer,String >();
        hm.put(1, "Ukraine"); hm.put(2, "Russia");
        hm.put(3, "EU"); hm.put(4, "NATO");
        // Displaying HashMap
        System.out.println(hm);
        // Traversing the map
        for(Map.Entry<Integer, String> entry : hm.entrySet()) {
        System.out.println(entry.getKey()+" : "+entry.getValue());
        }
        hm.remove(2);
        System.out.println("After Removing 2 :\n"+hm);

}}
```

Traversing and Removing Elements From HashMap+**Code and output**

```
1   package CollectionClass;
2   import java.util.*;
3   class TraverseHM {
4   public static void main(String args[]){
            HashMap< Integer,String> hm = new HashMap< Integer,String >();
6           hm.put(1, "Ukraine"); hm.put(2, "Russia");
7           hm.put(3, "EU"); hm.put(4, "NATO");
8           // Displaying HashMap
9           System.out.println(hm);
10          // Traversing the map
11          for(Map.Entry<Integer, String> entry : hm.entrySet()) {
12          System.out.println(entry.getKey()+" : "+entry.getValue());
13          }
14          hm.remove(2);
15          System.out.println("After Removing 2 :\n"+hm);
16
17  }}
```

```
run:
{1=Ukraine, 2=Russia, 3=EU, 4=NATO}
1 : Ukraine
2 : Russia
3 : EU
4 : NATO
After Removing 2 :
{1=Ukraine, 3=EU, 4=NATO}
```

Replace HashMap Elements

HashMap provides built-in methods to replace elements. There are two overloaded replace methods:

1. First method takes two arguments one for key and second for the value we want to replace with.

```
hm.replace(1, " Ukraine ");
```

2. Second method takes three arguments first is key and second is value associated with the key and third is value that we want to replace with the key-value.

```
hm.replace(1, " Ukraine ", "Russia");
```

Replacing an Elements From HashMap+Code and output

```
import java.util.*;

class replaceEHM {
public static void main(String args[]){
        HashMap< Integer,String> hm = new HashMap< Integer,String >();
        hm.put(1, "Ukraine"); hm.put(2, "Russia");
        hm.put(3, "EU"); hm.put(4, "NATO");hm.put(5, "China");
        // Displaying HashMap
        System.out.println(hm);
        // Replacing element from the map
        hm.replace(1, "Innocent suffering");
        System.out.println("After replacing element 1 :\n"+hm);

}}
```

```
run:
{1=Ukraine, 2=Russia, 3=EU, 4=NATO, 5=China}
After replacing element 1 :
{1=Innocent suffering, 2=Russia, 3=EU, 4=NATO, 5=China}
```

TreeMap Class

TreeMap Class

is a implementation of Map interface based on red-black tree. It provides an efficient way of storing key-value pairs in sorted order.

This class is similar to HashMap class except **it is sorted in the ascending order of its keys**. It is not suitable for thread-safe operations due to its unsynchronized nature.

TreeMap class extends AbstractMap class and implements NavigableMap interface .

Declaration.

```
public class TreeMap<K,V>extends AbstractMap<K,V>implements NavigableMap<K,V>, Cloneable, Serializable
```

TreeMap Class+

Important Features:

1. It contains only unique elements.
2. It cannot have a null key but can have multiple null values.
3. It is non synchronized.
4. It maintains ascending order.

TreeMap Constructors

Constructor	**Description**
TreeMap()	creates a new, empty tree map, using the natural ordering of its keys.
TreeMap(Comparator<? super K> comparator)	created a new, empty tree map, ordered according to the given comparator.
TreeMap(Map<? extends K,? extends V> m)	creates a new tree map containing the same mappings as the given map.
TreeMap(SortedMap<K,? extends V> m)	creates a new tree map containing sortedmap.

TreeMap Class Methods+

The table below contains the methods of TreeMap that can be used to manipulate objects.

Method	**Description**
SortedMap<K,V> headMap(K toKey)	returns the key-value pairs whose keys are strictly less than toKey.
NavigableMap<K,V> headMap(K toKey, boolean inclusive)	returns the key-value pairs whose keys are less than (or equal to if inclusive is true) toKey.
Map.Entry<K,V> higherEntry(K key)	returns the least key strictly greater than the given key, or null if there is no such key.
K **higherKey**(K key)	is used to return true if this map contains a mapping for the specified key.
Set **keySet**()	returns the collection of keys exist in the map.

TreeMap Class Methods++

The table below contains the methods of TreeMap that can be used to manipulate objects.

Method	Description
NavigableSet<K> descendingKeySet()	returns a reverse order NavigableSet view of the keys contained in the map.
NavigableMap<K,V> descendingMap()	returns the specified key-value pairs in descending order.
Map.Entry **firstEntry**()	returns the key-value pair having the least key.
Map.Entry<K,V> **floorEntry**(K key)	returns the greatest key, less than or equal to the specified key, or null if there is no such key.
void **forEach**(BiConsumer<? super K,? super V> action)	performs the given action for each entry in the map until all entries have been processed or the action throws an exception.

TreeMap Class Methods+++

The table below contains the methods of TreeMap that can be used to manipulate objects.

Method	Description
Map.Entry<K,V> **lastEntry**()	It returns the key-value pair having the greatest key, or null if there is no such key.
Map.Entry<K,V> **lowerEntry**(K key)	It returns a key-value mapping associated with the greatest key strictly less than the given key, or null if there is no such key.
K **lowerKey**(K key)	It returns the greatest key strictly less than the given key, or null if there is no such key.
NavigableSet **navigableKeySet**()	It returns a NavigableSet view of the keys contained in this map.
Map.Entry<K,V> **pollFirstEntry**()	It removes and returns a key-value mapping associated with the least key in this map, or null if the map is empty.

TreeMap Class Methods++++

The table below contains the methods of TreeMap that can be used to manipulate objects.

Method	**Description**
Map.Entry<K,V> pollLastEntry()	removes and returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
V **put**(K key, V value)	inserts the specified value with the specified key in the map.
void **putAll(Map**<? extends K,? extends V> map)	is used to copy all the key-value pair from one map to another map.
V **replace**(K key, V value)	replaces the specified value for a specified key.
boolean **replace**(K key, V oldValue, V newValue)	replaces the old value with the new value for a specified key.

TreeMap Class Methods+++++

The table below contains the methods of TreeMap that can be used to manipulate objects.

Method	**Description**
boolean **containsKey**(Object key)	It returns true if the map contains a mapping for the specified key.
boolean **containsValue**(Object value)	It returns true if the map maps one or more keys to the specified value.
K **firstKey**()	It is used to return the first (lowest) key currently in this sorted map.
V **get**(Object key)	It is used to return the value to which the map maps the specified key.
K **lastKey**()	It is used to return the last (highest) key currently in the sorted map.

TreeMap Class Methods++++++

Method	Description
void **replaceAll**(BiFunction<? super K,? super V,? extends V> function)	It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
NavigableMap<K,V> **subMap**(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)	It returns key-value pairs whose keys range from fromKey to toKey.
SortedMap<K,V> **subMap**(K fromKey, K toKey)	It returns key-value pairs whose keys range from fromKey, inclusive, to toKey, exclusive.
SortedMap<K,V> **tailMap**(K fromKey)	It returns key-value pairs whose keys are greater than or equal to fromKey.
NavigableMap<K,V> **tailMap**(K fromKey, boolean inclusive)	It returns key-value pairs whose keys are greater than (or equal to, if inclusive is true) fromKey.

TreeMap Class Methods+++++++

The table below contains the methods of TreeMap that can be used to manipulate objects.

Method	**Description**
V **remove**(Object key)	removes the key-value pair of the specified key from the map.
int **size**()	returns the number of key-value pairs exists in the hashtable.
Collection **values**()	returns a collection view of the values contained in the map.

Example : Creating and Adding Elements to TreeMap

In this example, we are creating TreeMap, then add elements to it.

We used **put()** method to insert elements which takes two arguments:

first is key and **second is value**.

Example : Creating and Adding Elements to TreeMap

```
import java.util.*;

class TMPro {
public static void main(String args[]){
      TreeMap<String,Integer> tm = new TreeMap<String,Integer>();
      tm.put("A",100); tm.put("B",200);
      tm.put("C",344); tm.put("D",400);
      // Displaying TreeMap
      System.out.println(tm);
}}
```

Note the use of put() method in the insertion of the elements into the TreeMap.

Example : Creating and Adding Elements to TreeMap+code and output

```
1   package CollectionClass;
2   import java.util.*;
3   class TMPro {
4   public static void main(String args[]){
            TreeMap<String,Integer> tm = new TreeMap<String,Integer>();
6           tm.put("A",100); tm.put("B",200);
7           tm.put("C",344); tm.put("D",400);
8           // Displaying TreeMap
9           System.out.println(tm);
10  }
11  }
```

```
run:
{A=100, B=200, C=344, D=400}
```

Fetch first and last key in TreeMap

We can get first and last element of the map using the **firstEntry()** and **lastEntry()** method. It returns a pair of key and value.

```
import java.util.*;

class fetchTMElements {
public static void main(String args[]){
      TreeMap<String,Integer> tm = new TreeMap<String,Integer>();
      tm.put("A",100); tm.put("B",200);
      tm.put("C",344); tm.put("D",400);
      // Displaying TreeMap
      System.out.println(tm);
      // First Element
      System.out.println("First Element: "+tm.firstEntry());
      // Last Element
      System.out.println("Last Element: "+tm.lastEntry());
}}
```

Fetch first and last key in TreeMap+code and output

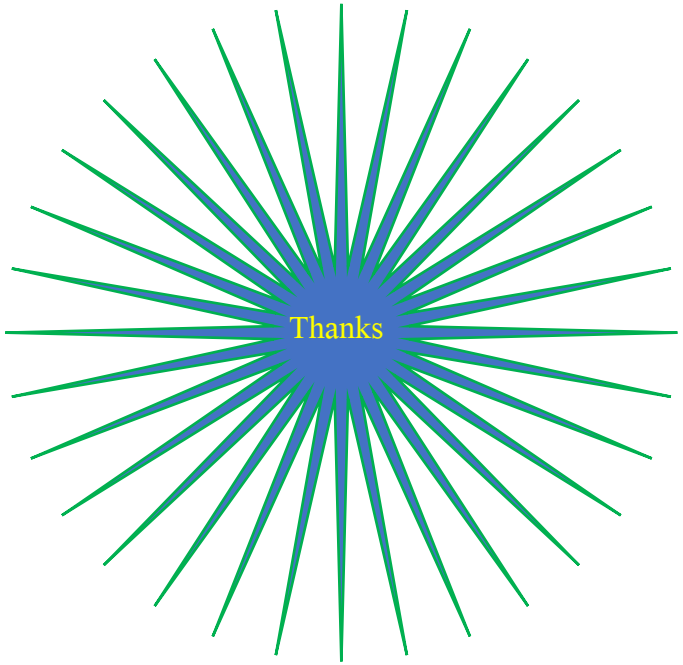
```
1   package CollectionClass;
2   import java.util.*;
3
4   class fetchTMElements {
5   public static void main(String args[]){
6           TreeMap<String,Integer> tm = new TreeMap<String,Integer>();
7           tm.put("A",100); tm.put("B",200);
8           tm.put("C",344); tm.put("D",400);
9           // Displaying TreeMap
10          System.out.println(tm);
11          // First Element
12          System.out.println("First Element: "+tm.firstEntry());
13          // Last Element
14          System.out.println("Last Element: "+tm.lastEntry());
15  }}
```

```
run:
{A=100, B=200, C=344, D=400}
First Element: A=100
Last Element: D=400
```

Assignment

1. If We can get head elements and tail elements of the TreeMap by using the headMap() and tailMap() methods, Write a program that will fetch the head and tail elements of a given TreeMap.
2. Find out how to iterate a TreeMap.

Thank you for Listening



References

Java Collection Framework treeset. Studytonight.com. (n.d.). Retrieved November 12, 2022, from https://www.studytonight.com/java/treeset-in-collection-framework.php

Accessing a Java collection using iterators. Studytonight.com. (n.d.). Retrieved November 9, 2022, from https://www.studytonight.com/java/iterator-in-collection.php

Java Collection Framework Hashtable. Studytonight.com. (n.d.). Retrieved November 10, 2022, from https://www.studytonight.com/java/hashtable-in-collection-framework.php

Java Collection Framework Linked List. Studytonight.com. (n.d.). Retrieved November 9, 2022, from https://www.studytonight.com/java/linkedlist-in-collection-framework.php

Java Collection Framework treemap. Studytonight.com. (n.d.). Retrieved November 10, 2022, from https://www.studytonight.com/java/treemap-in-collection-framework.php

Map interface - java collections. Studytonight.com. (n.d.). Retrieved November 11, 2022, from https://www.studytonight.com/java/map-interface-in-java.php

Accessing a Java collection using iterators. Studytonight.com. (n.d.). Retrieved November 9, 2022, from https://www.studytonight.com/java/iterator-in-collection.php