Object Oriented Programming with Java (Subject Code: BCS-403)

Unit 4

Lecture 32

Lecture 32

- **SortedSet Interface**
- **TreeSet,**

SortedSet Interface

The SortedSet interface in Java extends the Set interface and provides a set that maintains its elements in ascending order. It is part of the java.util package and is primarily implemented by the TreeSet class.

Ordered Elements: The elements in a SortedSet are maintained in sorted order, either by their natural ordering (if they implement the Comparable interface) or by a Comparator provided at the time of set creation.

No Duplicates: Like other Set implementations, a SortedSet does not allow duplicate elements.

Null Elements: In general, a SortedSet does not allow null elements. However, some implementations like TreeSet allow one null element.

Implementation: The primary implementation of the SortedSet interface is the TreeSet class, which is backed by a self-balancing binary search tree (Red-Black Tree).

Sorting and Searching: Because the elements are stored in sorted order, operations like retrieving the smallest, largest, or n-th element, as well as searching for elements, can be performed efficiently.

Iterators and Subsets: The SortedSet interface provides methods for retrieving iterators that traverse the set in sorted order. It also allows you to retrieve subsets of the set based on a range of elements.

Example

```
import java.util.SortedSet;
import java.util.TreeSet;
 public class SortedSetExample {
public static void main(String[] args) {
// Creating a SortedSet (TreeSet)
 SortedSet<Integer> numbers = new TreeSet<>();
 // Adding elements to the SortedSet
 numbers.add(5);
numbers.add(2);
numbers.add(8);
numbers.add(1);
numbers.add(3);
 System.out.println("SortedSet of numbers: " + numbers);
// Output: SortedSet of numbers: [1, 2, 3, 5, 8]
```

Example

```
// Retrieving the first and last elements
System.out.println("First element: " + numbers.first());
// Output: First element: 1
System.out.println("Last element: " + numbers.last());
 // Output: Last element: 8
 // Retrieving a subset
SortedSet<Integer> subset = numbers.subSet(2, 5);
System.out.println("Subset from 2 to 5: " + subset);
 // Output: Subset from 2 to 5: [2, 3]
// Iterating over the SortedSet
System.out.print("Iterating over the SortedSet: ");
 for (Integer number : numbers)
{ System.out.print(number + " "); }
 System.out.println();
// Output: Iterating over the SortedSet: 1 2 3 5 8 } }
```

In the above example, we create a SortedSet (implemented by TreeSet) of Integer elements. We demonstrate adding elements to the set, retrieving the first and last elements, obtaining a subset based on a range, and iterating over the set.

Notice that the elements are automatically sorted in ascending order.The SortedSet interface is useful when you need to maintain a sorted collection of unique elements. It provides efficient methods for retrieving elements based on their sorted position, as well as for obtaining subsets of the set within a specific range.

One common use case for SortedSet is when you need to perform operations that rely on the sorted order of elements, such as finding the smallest or largest element, or retrieving elements within a specific range.

TreeSet

TreeSet is a concrete implementation of the SortedSet interface in Java. It is a self-balancing binary search tree implementation, specifically a Red-Black tree. Elements in a TreeSet are stored in ascending order according to their natural ordering or a Comparator provided at set creation time.

Ordering: TreeSet stores its elements in sorted ascending order. If the elements are of a class that implements the Comparable interface, they are sorted according to their natural ordering. If not, you can provide a custom Comparator at the time of set creation to define the sorting order.

No Duplicates: Like other Set implementations, TreeSet does not allow duplicate elements. Attempts to add a duplicate element will be ignored.

TreeSet

Null Handling: By default, TreeSet does not allow null elements. If you try to add a null element, it will throw a NullPointerException. However, it is possible to have one null element in a TreeSet by providing a custom Comparator that handles null values appropriately.

Self-Balancing: TreeSet is implemented as a self-balancing binary search tree (Red-Black tree). This means that after every insertion or deletion operation, the tree is rebalanced to maintain its height within reasonable limits, ensuring logarithmic time complexity for operations like add(), remove(), and contains().

TreeSet

Iterators and Subsets: TreeSet provides methods for retrieving iterators that traverse the set in sorted order. It also allows you to retrieve subsets of the set based on a range of elements.

Performance: The time complexity for basic operations like add(), remove(), and contains() is O(log n) on average, where n is the number of elements in the set. This makes TreeSet efficient for large datasets.

Example

```
import java.util.TreeSet;
 public class TreeSetExample {
public static void main(String[] args) {
// Creating a TreeSet
TreeSet<Integer> numbers = new TreeSet<>();
// Adding elements to the TreeSet numbers.add(5);
numbers.add(2);
 numbers.add(8);
numbers.add(1);
numbers.add(3); System.out.println("TreeSet of numbers: " +
numbers);
 // Output: TreeSet of numbers: [1, 2, 3, 5, 8]
```

Example

```
// Retrieving the first and last elements
System.out.println("First element: " + numbers.first());
 // Output: First element: 1
 System.out.println("Last element: " + numbers.last());
// Output: Last element: 8
// Retrieving a subset
TreeSet<Integer> subset = (TreeSet<Integer>) numbers.subSet(2, 5);
 System.out.println("Subset from 2 to 5: " + subset);
// Output: Subset from 2 to 5: [2, 3] //
Iterating over the TreeSet
System.out.print("Iterating over the TreeSet: ");
for (Integer number : numbers) { System.out.print(number + " "); }
 System.out.println(); // Output: Iterating over the TreeSet: 1 2 3 5 8 }
}
```

In this example, we create a TreeSet of Integer elements. We demonstrate adding elements to the set, retrieving the first and last elements, obtaining a subset based on a range, and iterating over the set. Notice that the elements are automatically sorted in ascending order.TreeSet is useful when you need to maintain a sorted collection of unique elements and perform operations that rely on the sorted order of elements, such as finding the smallest or largest element, or retrieving elements within a specific range.

It provides efficient operations due to its self-balancing binary search tree implementation.

However, keep in mind that the insertion, removal, and retrieval operations in a TreeSet are more expensive than those in a HashSet because of the overhead of maintaining the sorted order. If you don't need the elements to be sorted, a HashSet may be a better choice for better performance.