Write a Program to Insert String Objects into the TreeSet where the Sorting Order is of

Reverse of Alphabetical Order:

Eg: Sorting\_String\_Objects\_InAlphabetical\_Order\_Using\_TreeSet

// go through the code

Eg: Sorting\_String\_Objects\_InReverse\_Order\_Using\_TreeSet

// go through the code

Write a Program to Insert StringBuffer Objects into the TreeSet where Sorting Order is Alphabetical Order:

Eg: Sorting\_StringBuffer\_Objects\_InAlphabetical\_Order\_Using\_TreeSet

In jdk 17 StringBuffer implements Comparable so we get sorted order

In previous versions of jdk StringBuffer does not implement Comparable

Eg: Sorting\_StringBuffer\_Objects\_InReverse\_Order\_Using\_TreeSet

Write a Program to Insert String and StringBuffer Objects into the TreeSet where Sorting Order is Increasing Length Order.

If 2 Objects having Same Length then Consider their Alphabetical Order:

eg: A,ABC,AA,XX,ABCE,A

Eg: String\_StringBuffer\_Objects\_Sorting\_In\_Increasing\_Length

// construct the binary tree

Note :

If we are using TreeSet() , then the condition is

1. Object should be homogenous
2. Object should be comparable ( class should implement Comparable(I))

If we are using TreeSet(Comparator c ) then the condition is

1. Object need not to be homogenous ( like the above example)
2. Object need not to implement comparable

When to go for Comparable interface and when to go for Comparator interface?

For predefined Comparable classes ( like String ) default natural sorting order is already available . if we are not satisfied with that we can define our own sorting by Comparator object

For predefined Non-Comparable classes ( like StringBuffer in jdk 11) default natural sorting is not already available . if we want to define our own sorting we can use comparator object .

For our own Classes (like Employee) the person who is writing the Employee class he is responsible to define his own sorting order by implementing Comparable interface

The person who is using our own class if he is not satisfied with our default natural sorting order he can define his own sorting order by using Comparator Object

If he is satisfied with the default natural sorting order then he can use our class directly

Write a Program to Insert Employee Objects into the TreeSet where default natural sorting order is Based on Ascending Order of Employeeld and

Customized Sorting Order is Based on Alphabetical Order of Names:

Eg: Customized\_Objects\_InTreeSet\_Based\_On\_Comparable\_Compartor

Comparable(I)

Present in java.lang Package

It is Meant for Default Natural Sorting Order.

Defines Only One Method compareTo()

All Wrapper Classes and String Class implements Comparable Interface.

Comparator(I)

Present in java.util Package

It is Meant for Customized Sorting Order.

Defines 2 Methods compare() and equals().

The Only implemented Classes of Comparator are Collator and RuleBaseCollator.

Functional interfaces :

If an interface contains only one abstract method then it is called functional interface

Javap java.util.function.Predicate;

public interface java.util.function.Predicate<T> {

public abstract boolean test(T);

// functional interface

// default methods available for utility methods for devevloper.

public default java.util.function.Predicate<T> and(Predicate p);

public default java.util.function.Predicate<T> negate();

public default java.util.function.Predicate<T> or(Predicate p );

// the above three are concrete methods , body of these methods is available in the Predicate interface , here public is just the useful for the visibility.

public static <T> java.util.function.Predicate<T> isEqual(java.lang.Object);

public static <T> java.util.function.Predicate<T> not(Predicate p );

// static method body is already available .

}

Eg: Predicate\_Test\_Method

Eg: Predicate\_Test\_Method\_Without\_Lambda\_Expression

Eg: Predicate\_Test\_Method\_With\_Lambda\_Expression

// use of test() method .

Take a particular input for that input you make a test give some results , for that purpose we use this method of predicate class.

Eg: Array\_With\_Predicate\_Test\_Method\_Eg1

// go through the code

Eg: Array\_With\_Predicate\_Test\_Method\_Eg2

// go through the code

Eg: Array\_With\_Predicate\_Test\_Method\_Eg3

// go through the code

Eg: Array\_With\_Predicate\_Test\_Method\_Eg4

// go through the code

Eg: Array\_With\_Predicate\_Test\_Method\_Eg5

// go through the code

Function (I) :

public interface java.util.function.Function<T, R> {

// functional interface

public abstract R apply(T);

T- Input type

R - return type

public default <V> Function <V,R> compose(Function<? super V, ? extends T>);

public default <V> Function<T, V> andThen( Function<? super R, ? extends V>);

public static <T> Function<T, T> identity();

}

Eg: Function\_Apply\_Method

Eg: Function\_Apply\_Using\_lambda

When to go for Predicate and when to go for Function?

Predicate -> To implement some conditional checks we should go for Predicate

Function -> To perform some operation and return some result we should go for Function

Method(::)Reference and constructor(::)Reference

========================================

:: -> scope resolution operator

Syntax for method reference

========================

1. static method

Classname::methodName

1. instance method

object::methodName

// first check this example before going to method reference

Eg: Thread\_With\_Lambda\_Expression

Eg: Method\_Reference\_Using\_Static\_Method

Eg: Method\_Reference\_Using\_Instance\_Method