Set [Interface]: TreeSet

Definition

The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided. The implementation provides guaranteed log(n) time cost for the basic operations which are add, remove, and contains.

A treeSet instance performs all element comparisons using its compareTo method, so two elements that are deemed equal by this method are, from the standpoint of the set, equal.

This implementation is not synchronized, if multiple thread accesses a tree set concurrently, and at least one of the threads modifies the set, it must be synchronized externally.

The TreeSet is an implementation of self-balancing binary search tree, operation takes 0(log(N)) time. The self-balancing tree makes sure that the height of the tree is always 0(log(N)) for all operations. This is one of the most efficient data structures in order to store the huge sorted data and performs operations on it.

Constructor summary

TreeSet ()	Constructs an empty TreeSet Object in which elements will get stored in default natural sorting order.
TreeSet(Comparator)	Constructs an empty TreeSet Object in which elements will need an external specification of the sorting order.
TreeSet(Collections)	Constructs TreeSet object containing all the elements from the giving collection in which element get stored in default natural sorting order.
TreeSet(SortedTest)	Constructs an treeSet Object containing all the elements from the given sortetest in which element get stored in default natural sorting order.

Methods summary

Modifier and Type	**Method and Description**
boolean	**add**(**E** e) Adds the specified element to this set if it is not already present.
void	**Clear**() Removes all elements from this set
boolean	**addAll**(Collection < ? extends E > c) Appends all of the elements in the specified collections to the end of this set.
Comparator< ? super **E**>	**comparator**()

	Returns the comparator used to order the elements in this set, or null if this set uses the natural ordering of its elements.
boolean	**contains** (Object o) Returns true if this set contains the specified element
Iterator<E>	**iterator** ( ) Returns an iterator over the elements in this set in descending order.
E	**first**() returns the first (lowest) element in the current set.
E	**floor**(**E** e) Return the greatest element in this set less than or equal to the given element, or null if there is no such element.
NavigableSet<E>	**headSet**(E toElement, boolean inclusive) Returns a view of the portion of this set whose elements are less than (or equal to, if inclusive is true) toElement.
E	**higher**(**E** e) Returns the least element in this set strictly greater than the given element, or null if there is no such element
boolean	**isEmpty**() Returns true if this set contains no elements.
E	**last**() Returns the last (highest) element currently in this set.
E	**lower**(**E** e) Returns the greatest element in this set strictly less than the given element, or null if there is no such element.

When is its use recommended?

- When need to perform read/write operations frequently
- TreeSet has greater locality. If two entries are nearby in the order, TreeSet place them near each other in data structure and hence in memory.
- Sorted unique elements are required instead of unique elements. The sorted list is always in ascending order.