Lab III: Sets

(Due on February 27th, 2015)

A *set* is a collection of distinct elements.

$\{2, 3, 5, 7\}$ $\{$`'a'`, `'b'`, `'c'`$\}$ $\{$`"aardvark"`, `"aardwolf"`, `"albatross"`$\}$

We will represent sets as an ADT, but restrict ourselves to only sets whose elements can be *ordered*.

1 Sorted Set Abstract Data Type

The sorted sets you will represent here are similar to the sets you have studied in mathematics, but with two important restrictions:

1. the set contains only elements of one type (ex: `Integer`, `String`, etc…).
2. the elements can be ordered, meaning that there must be operations <, >, =, etc…defined for any type used in a sorted set.

Here are the operations available on sorted sets:

Empty/Full. Determine if the set is empty ($\{\}$) or full. A set is full if there is insufficient space to store additional elements.

Contains. The fundamental operation *contains* (denoted by $\in$) on a set is to tell if an element is contained within it:

$$a \in \{a, b, c\} \qquad c \notin \{a, b\}$$

Minimum/Maximum. Since the elements of a set are ordered, it is possible to determine the smallest and largest element in the set.

Add. Sorted sets are built using an operation for adding an element to an existing set. If we have a set $\{a, b\}$, then **add**(c) would yield $\{a, b, c\}$. Add should behave like the traditional set *union* operation $\cup$, in that adding a duplicate to a set would leave the set unchanged. Ex:

$$\{a, b, c\} \cup \{a\} = \{a, b, c\}$$

Remove. An element can be removed from a sorted set using the remove operation. If we have a set $\{a, b, c\}$, then **remove**(c) would yield $\{a, b\}$. Remove should be have like the traditional set *difference* operation $-$, in that removing an element not in the original set results in the set unchanged. Ex:

$$\{a, b, c\} - \{d\} = \{a, b, c\}$$

Traversal. A *traversal* of a set is a series of operations that enable a visiting of each of the elements in the set. For example the traversal of $\{1, 2, 3\}$ will be:

$$1 \longrightarrow 2 \longrightarrow 3$$

The traversal operation is broken down into 3 sub-operations: **reset**, **hasNext** and **next**. Resetting restarts the traversal, say after a previous traversal. Next gives the next element in the traversal and **hasNext** indicates the end.

1.1 Sorted Set API

This specification uses a type parameter `T`.

Prototype	`boolean contains(T element)`
Purpose	Determine if the sorted set contains an element.
Pre-conditions	None.
Post-conditions	If the element is found in the set that equals `element`, then returns `true`, otherwise, the method returns `false`.

Prototype	`boolean add(T element)`
Purpose	Add an element into the sorted set.
Pre-conditions	The sorted set is not full.
Post-conditions	If there is no element in the sorted set that equals `element`, then the element is inserted into the sorted set at its correct position according to the set's sort order. Otherwise the set is unchanged. The method returns `true` if the element is added to the set, and `false` otherwise.

Prototype	`boolean remove(T element)`
Purpose	Remove an element from the sorted set.
Pre-conditions	
Post-conditions	If there is an element in the sorted set equal to `element`, then it is removed from the sorted set. Otherwise, the sorted set is unchanged. The method returns `true` if the element is added to the set, and `false` otherwise.

Prototype	`int size()`
Purpose	Determine the number of elements in the sorted set.
Pre-conditions	None.
Post-conditions	Returns the number of elements in the sorted set.

Prototype	`boolean isEmpty()`
Purpose	Determine if the sorted set is empty.
Pre-conditions	None.
Post-conditions	Returns `true` if the sorted set is empty, `false` otherwise.

Prototype	`boolean isFull()`
Purpose	Determines if the sorted set is full.
Pre-conditions	None.
Post-conditions	Returns `true` if the sorted set is full, `false` otherwise.

Prototype	`T min()`
Purpose	Retrieve the smallest element in the set.
Pre-conditions	The sorted set is not empty.
Post-conditions	Returns the smallest element in the set. The set is unchanged.

Prototype	`T max()`
Purpose	Retrieve the largest element in the set.
Pre-conditions	The sorted set is not empty.
Post-conditions	Returns the largest element in the set. The set is unchanged.

Prototype	`void reset()`
Purpose	Initialize a traversal of the sorted set.
Pre-conditions	
Post-conditions	If the set contains elements, the traversal cursor is positioned on the first element. Otherwise, the traversal is complete (trivially).

Prototype	`boolean hasNext()`
Purpose	Determine if a traversal can continue.
Pre-conditions	The traversal has been initialized and no **add** or **remove** operations have been performed since the initialization.
Post-conditions	Returns `true` is there are elements left in the traversal, `false` otherwise.

Prototype	`T next()`
Purpose	Return the current element in the traversal, and then advance the traversal cursor to the next element in the set.
Pre-conditions	The traversal has been initialized and no **add** or **remove** operations have been performed since the initialization. The traversal still has at least one element left.
Post-conditions	If there is a next element, the traversal cursor has advanced to it and it is returned. At the end of the traversal cursor is *undefined*, meaning that is no longer refers to an element.

2 Sorted Set Data Structure

Implement the sorted set Data Structure as a generic class `SortedSet<T>`. Your implementation must use an array to store the elements. In order to sort the set elements, we must put an constraint on the type arguments that can be supplied for `T`. We will allow only classes that implement the method `compareTo()`, by declaring the class as follows:

```
public class SortedSet<T extends Comparable> {
  ...
}
```

The method `x.compareTo(y)` works by comparing `x` with `y` and returning an `int` result. If the result is < 0 then `x` $<$ `y`. If the results is > 0 then `x` $>$ `y`. Finally, if the result is 0, then `x` $=$ `y`.

Start from the supplied Java file, `SortedSet.java`, which contains the declaration of the `SortedSet<T>` class, as well as the constructor and an implementation of `contains()`. For each method, use the fact that the elements can be stored in sorted order to optimize the efficiency of your code. As an example, the `contains()` uses binary search instead of linear search. Other methods that can benefit include: `min()`, `max()` and `isSubset()`. You may change the private section of the class, but the public section should remain the same.

2.1 Unit Testing

Instead of testing your data structure using a `main()` method, use JUnit test cases to verify your implementation. Here is a list of tests that your class should be capable of passing:

Test Name	Test Description	Included
`testConstructorMakesEmptySet()`	The set is initialized to the empty set.	✓
`testContains()`	The operation **contains** works for the set.	✓
`testAdd()`	The operation **add** works for the set.	
`testNoDuplicates()`	Adding a duplicate elements to the set does not modify the set.	
`testFullSet()`	The operation **isFull** works for the set.	
`testFullSetException()`	Adding to a full set will cause an error.	
`testRemove()`	The operation **remove** works for the set.	
`testRemoveNonElement()`	The operation **remove** fails to remove a non-element of the set.	
`testEmptySet()`	The operation **isEmpty** works for the set.	
`testRemoveOnEmptySet()`	Removing from the empty set fails, but does not cause an error.	
`testMin()`	The operation **min** returns the minimum element in the set.	
`testMax()`	The operation **min** returns the minimum element in the set.	
`testMinMaxInSingletonSet()`	In a set with one element, the minimum and the maximum are the same.	
`testMinOnEmptySet()`	Calling **min** on an empty set gives an error.	✓
`testMaxOnEmptySet()`	Calling **max** on an empty set gives an error.	✓
`testTraversal()`	Test that a traversal visits each element in the set in turn.	✓
`testTraversalWithoutError()`	Test that a proper traversal does not cause an error.	✓
`testTraversalEmptySet()`	Test that an empty set has no elements in its traversal.	✓
`testAddDuringTraversal()`	Test that adding during a traversal causes an error.	✓
`testNextOnCompletedTraversal()`	Test that asking for more elements at the end of the traversal causes an error.	✓
`testIsSubset()`	Test that the **isSubset** operation works (bonus, see Section 5).	✓

3 Requirements

- In the `SortedSet<T>` class, implement the API methods from Section 1.1.
- The set elements must be stored in an array in sorted order (either ascending or descending).
- Your data structure should pass all tests In the `SortedSetTest` class. Make sure you add all the missing test from Section 2.1.
- Global variables are prohibited.

4 Hand-in Checklist

- ☐ The program is clear and well commented. All names follow Java convention.
- ☐ The project directory contains all of the source files, including the JUnit test class.
- ☐ Zip the project folder and submit it using Léa.

5 Bonus (15%)

Add the following operation to the sorted set:

Subset. Determine if every element of a set A is also in a second set B, that is, A is entirely in B. This is usually written as $A \subseteq B$. For example:

$$\{a, c\} \subseteq \{a, b, c, d\} \qquad \{b, d\} \not\subseteq \{a, b, c\}$$

A more formal definition of subset is: for all x, if $x \in A$ then $x \in B$.

Prototype	`boolean isSubset(SortedSet<T> rhs)`
Purpose	Determine if the current set is a subset of the provided set.
Pre-conditions	
Post-conditions	Returns `true` if all the elements of the current set are also in `rhs`, `false` otherwise. If the current set is empty, return `true`.

Implement the `isSubset()` method in the `SortedSet<T>` class. For full bonus marks, your implementation should only loop over the data once and not call any other methods of the `SortedSet<T>` class (except for `compareTo()`). Verify your implementation using the provided unit test.