Lab II: Sets

(Due on Check on Léa)

A *set* is a collection of distinct elements.

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

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

1 Set Abstract Data Type and API

The sets you will represent here are similar to the sets you have studied in mathematics, but with one important difference: the set contains only elements of one type (ex: `Integer`, `String`, etc…).

1.1 Operations

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

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

Contains all. Determine if every element of a set A is also in a second set B, that is, A is entirely in B (a.k.a. subset). 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 $x \in B$.

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:

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

Size. Number of elements of the set (a.k.a. cardinality). This is usually written as $|A|$.

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

To String. Operations to get a string representation of a set (see API).

1.2 Set API

This specification uses a type parameter `T`.

Prototype	`boolean contains(T element)`
Purpose	Determine if the set contains a specific 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 containsAll(Set<T> rhs)`
Purpose	Determine if the set contains all the elements of the provided set, or, that the provided set is a subset of the current set.
Pre-conditions	None.
Post-conditions	Returns `true` if all the elements of `rhs` are in the current set, `false` otherwise. If the provided set is empty, return `true`.

Prototype	`boolean add(T element)`
Purpose	Add an element into the set.
Pre-conditions	The set is not full.
Post-conditions	If there is no element in the set that equals `element`, then the element is inserted into the set, 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 set.
Pre-conditions	None.
Post-conditions	If there is an element in the set equal to `element`, then it is removed from the set. Otherwise, the 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 set.
Pre-conditions	None.
Post-conditions	Returns the number of elements in the set.

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

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

Prototype	`String toString()`
Purpose	Get a `String` representation of the set.
Pre-conditions	None.
Post-conditions	Returns a `String` representation of the set, consisting of: • a `{`, • the `String` representations of the elements, comma-separated, and • a `}`.

2 Sorted Set Abstract Data Type and API

A *sorted* set is a set that has a further restriction: the elements can be ordered, meaning that there must be operations <, >, =, etc…defined for any type used in a sorted set.

2.1 Operations

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

Subset. Produce a subset of the current set containing all the elements between a lower bound (inclusive) and an upper bound (exclusive). For example, the **subset**$(2, 4)$ of the set $\{1, 2, 3, 4, 5\}$ is $\{2, 3\}$.

2.2 Sorted Set API

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	`SortedSet<T> subset(T low, T high)`
Purpose	Produce a subset of the current set containing all the elements between `low` (inclusive) and `high` (exclusive)
Pre-conditions	The `low` must be less than or equal to `high`.
Post-conditions	Returns the subset.

3 Traversable Abstract Data Type and API

A *traversal* is a sequence of steps visiting each element in the data type.

Traversal. A *traversal* is a series of operations that enable a visiting of each of the elements in the collection. 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.

3.1 Traversable API

Traversals are used to "loop" over the data type. For example, if `data` is a traversable collection of integers, then we can write the following:

```
data.reset();
while(data.hasNext()) {
   int x = data.next();
   // code to run for each element
}
```

Prototype	`void reset()`
Purpose	Initialize a traversal.
Pre-conditions	
Post-conditions	If there are 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 modifications (ex: **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.
Pre-conditions	The traversal has been initialized and no modifications (**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.

4 Sorted Set Data Structure

Implement the above APIs as a `SortedSet` data structure. Your implementation must use an array to store the elements and keep them is a sorted order. The `Set` and `Traversable` APIs are included as Java *interfaces* to be implemented by your data structure.

The implementation will be a generic class, i.e.: `SortedSet<T>`. 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<T>> implements ... { ... }
```

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()`. You may add/remove private members of the class, but the public methods should remain the same.

5 Unit Testing

Instead of testing your data structure using a `main()` method, use JUnit test cases to verify your implementation. Appendix A has a list of tests provided in the starter that your class should pass.

6 Efficiency

For each `Set` and `SortedSet` API method, use the fact that the elements are be stored in sorted order to optimize the efficiency of your code. Hint: many methods can benefit from binary search, for example the `contains()` can use binary search instead of linear search.

7 Requirements

Your program *must* meet the following requirements:

1. Your program should be clear and well commented. It must follow the “420-406 Style Guidelines” (on Léa).
2. The `SortedSet<T>` class implements the API methods from Sections 1, 2 and 3.
3. The set elements must be stored in an array in sorted order (either ascending or descending). Do not use a list to store the data.
4. Use the fact that the elements can be stored in sorted order to optimize the efficiency of your code (Section **??**).
5. Do not do a complete sort of the array of elements. For example, no sorting using `Arrays.sort(..)`.
6. Your data structure should pass all tests in the `SortedSetTest` class.
7. Submit using Git by following the instructions (on Léa).

A Appendix: Provided JUnit test cases

Set API Tests.

Test Name	**Test Description**
`testConstructorMakesEmptySet()`	The set is initialized to the empty set.
`testContains()`	The operation **contains** works for the set.
`testContainsAll()`	Test that the **containsAll** operation works.
`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.
`testToArray()`	Converting the set into an array of elements gives expected results.
`testToString()`	Converting the set into its `String` representation gives expected results.

Sorted Set API Tests.

Test Name	**Test Description**
`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.
`testSubset()`	Test that the **subset** method generates proper subsets.
`testSubsetOnEmptySet()`	Test that the **subset** operation on an empty set returns an empty set
`testSubsetIllegalArguments()`	Test that when **subset** is called with `low` > `high` an error is thrown.

Traversal API Tests.

Test Name	Test Description
`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.