420-406-AB Data Structures
Winter 2020

February 10th
Worth 5%

Assignment 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 create a set data structure, but restrict ourselves to only sets whose elements can be *ordered*.

1 Sets

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:

$$\{a,c\} \subseteq \{a,b,c,d\} \qquad \{b,d\} \nsubseteq \{a,b,c\}$$

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`.

Signature	`boolean contains(T element)`
Pre-conditions	None.
Post-conditions	Determine if the set contains a specific element.
Returns	If the element is found in the set that equals `element`, then returns `true`, otherwise, the method returns `false`.

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

Signature	`boolean add(T element)`
Pre-conditions	The set is not full.
Post-conditions	Add an element into the set. If there is no element in the set that equals `element`, then the element is inserted into the set, otherwise the set is unchanged.
Returns	The method returns `true` if the element is added to the set, and `false` otherwise.

Signature	`boolean remove(T element)`
Pre-conditions	None.
Post-conditions	Remove an element from the set. If there is an element in the set equal to `element`, then it is removed from the set. Otherwise, the set is unchanged.
Returns	The method returns `true` if the element is added to the set, and `false` otherwise.

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

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

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

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

2 Sorted Sets

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

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

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

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

Signature	`void reset()`
Pre-conditions	None.
Post-conditions	Initialize a traversal.If there are elements, the traversal cursor is positioned on the first element. Otherwise, the traversal is complete (trivially).
Returns	None.

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

Signature	`T next()`
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.
Returns	Return the current element in the traversal, and then advance the traversal cursor to the next 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. These tests are meant to show you that your data structure is working, but passing them doesn't mean that your implementation is perfect.

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 Checklist" (in assignments directory on the GitHub repository).
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 6).

5. Do not do a complete sort of the array of elements. For example, no sorting using `Arrays.sort( .. )`.
6. Storing the elements in a list, even temporarily, is prohibited, since it is likely less efficient than working with the array directly.
7. Your data structure should pass all tests in the `SortedSetTest` class.
8. Submit using Git by following the instructions (in assignments directory on the GitHub repository).