Homework #1 – Implementing Set with a Linked List

Project Objectives

1. Be able to integrate the knowledge of Java Generics, Collection Framework and the Linked List data structure to implement the Set ADT. Components emphasized are:
    - Allowing parameterized type for handling a general class of values in the collection framework;
    - Understanding conceptual differences between lists and sets and implementing them with methods;
    - Implementation of an iterator with functionality that is appropriate for the collection, namely the set;
    - Using iterator for clean implementation of other methods of the collection (Set).
2. Be able to consider alternative implementation of the underlying data structure and understand the behavioral changes that may result. Following two versions of linked list data structure are to be used:
    - Singly linked list with a check to avoid repetition of an existing element—new elements are added in front of list;
    - Self-organizing singly linked list, also with the check to avoid repetition of elements—but with a Move-to-Front (MTF) strategy which removes an element from its current position on every successful search (including the one to avoid repetition while adding) and adds to the front.
3. Develop driver/test code to test correct implementation.

Deliverables

Screen capture of your programs' result and zipped Java source code for the following:

1. Class implementation of `Set<E>` interface using ordinary linked list (`LinkedSet`)
2. Class implementation of `Set<E>` interface using self-organized (MTF) linked list (`SOLinkedSet`)
3. One or more test/driver class with main method for testing the above two classed

The Problem

Developing an alternative to classes `TreeSet<E>` or `HashSet<E>` in Java Collection Framework for implementing the `Set<E>` interface (with limited features) using Singly Linked List as the underlying data structure.

Instructions:

Before you start writing code, please do the following:

- Understand how Java compiler looks up its library to find existing classes and interfaces. This includes components of the Java Collection Framework, all the relevant exceptions we use, among other things. While we can explicitly name the classes in the import statements at the beginning of our Java source files, most programmers add a blanket import

```
import java.util.*;
```

to get rid of the compilation errors related to missing symbols. As this programming assignment asks implementation for only a limited number of methods of the interface `Set<E>`, your completed work will still fall short of overriding all mandatory interfaces as defined in `java.util`., causing compilation errors. To alleviate this problem, download the attached `Set.java` file in your working directory and have it compiled. The local definition of `Set<E>` will supersede the definition from the `java.util` package.

- Understand Java Generics. You have had exposure to using generics from Lab#4. To develop an understanding of how to build a class for collection of generic elements, compare the source code in `LinkedList1.java` and `LinkedList2.java`. The former contains the source code for the class `LinkedList1` from the textbook chapter 19 (posted in Module 3), showing how to implement a singly linked list for only String objects. The latter shows the same code adapted to create the class `LinkedList2<E>` for a generic type `E`.
- Understand how iterators are implemented. You have had exposure to working with iterators from Lab#5. To understand how iterators are implemented, compare the `LinkedList2.java` and `LinkedList3.java`. The generic linked list implementation is now augmented with an iterator, adding the nested `Iterator<E>` class and the `iterator` method for the list. Here are a few things to note:
    - While the list methods of `LinkedList3` could have been rewritten using an iterator, they are left unaltered for easy comparison with the `LinkedList2`.
    - Having the iterator implemented as private nested class within `LinkedList3` has the same capability the other private nested node class has, i.e., they get direct access to the private data fields. Moreover, code residing in an outside class (e.g., main method in a driver class) cannot have an iterator created for `LinkedList3` instance `myList` as follows:

      ```
      Iterator<E> it = myList.new LisnkedListIterator();
      ```

      This results in a compilation error due to private access specifier for the inner class. The right way to obtain a new instance of iterator is using the public iterator method in `LinkseList3`, as in

      ```
      Iterator<E> it = myList.iterator();
      ```

Task #1. Develop the class **`LinkedSet<E>`**

Assuming you already have the file `Set.java` saved in your working directory, open it and look for all the methods you need to implement the `Set<E>` interface. Develop your `LinkedSet<E>` class in a file named `LinkedSet.java` with the prototype

```
public class LinkedSet<E> implements Set<E>
```

Your class should consist of the following members. Decide upon the access specifier to use for each of these members, including the nested classes. Use the linked list class examples as guidelines, but the recommendation is that you develop the iterator class first and implement the set interface methods using the iterator methods.

- A nested node class for storing elements of type E and reference to the next node
    - Data fields (note that the outer class has direct access to inner class data)
    - Method, including constructors
- A nested iterator class implementing the iterator interface

  ```
  private class LinkedSetIterator implements Iterator<E>
  ```

(look at the Java documentation for the iterator interface in `java.util` package).
    - Carefully choose the data fields. How many reference variables are needed?
    - Decide upon whether to make constructors public—this depends on how you plan on instantiating an iterator.
    - Methods enforced by the iterator interface
        - `hasNext` to check if there is any more element
        - `next` to return the next element
        - `remove` to remove the element returned by last call to `next`
- Data fields needed for the whole set object
- Methods enforced by the set interface (as per downloaded `Set.java`)
    - `clear` to remove all elements from the set
    - `isEmpty` to check if the set has no elements
    - `size` to find the number of elements in the set
    - `iterator` to get a new instance of set iterator
    - `contains` to check if a given element is in the set
    - `add` to check if a given element is not in the set and subsequently added (new elements are always added to the front of the underlying linked list)
    - `remove` to check if a given element is in the set and subsequently removed
- Other optional method you may find useful
    - `toString` to print the whole set (element by element printing can be easily done using the iterator, which we already have).

Recall that the implementation of many of the above set methods get easy if you think of traversing the underlying linked list implementation of the set using an iterator.

Task #2. Develop the class `SOLinkedSet<E>`

Once the previous task is done, consider the alternative implementation of the set by using a self-organized linked list as the underlying data structure. A self-organized list differs from the ordinary list in that it does guarantee preserving any explicit order of elements in a list. Instead, when an element is searched for and found in the list, it is removed from its current position and added to the front of the list. The motivation comes from the fact that a recently accessed element is more likely to be accessed again, and this move-to-front (MTF) strategy will have the frequently used elements migrated towards the front of the list, resulting in lower search time.

Develop the `SOLinkedSet<E>` class in a file named `SOLinkedSet.java` with the prototype

```
public class SOLinkedSet<E> implements Set<E>
```

which should also implement the `Set<E>` interface. In fact, the only methods which should change from `LinkedSet<E>` class (apart from the obvious class name change) due to the MTF strategy are

- `contains`, if the search for the element succeeds
- `add`, if the element is already there

In both cases, the element needs to be removed from its current position and then added to the front of the underlying linked list.

Task #3. Develop driver class(es) to test your code

You are free to choose how you develop your tester/driver code for both classes `LinkedSet<E>` and `SOLinkedSet<E>` (as a single driver class for each or one for both classes together). However, you must test that

- Set methods `add` and `remove` work as expected for both implementations. Repeated attempts to add the same element should change the order of elements stored (and printed) between the two implementations.
- Set query `contains` should not change the order of elements in ordinary linked list implementation, but searched elements should move to front in the self-organized list implementation.
- Set queries `isEmpty` and size are consistent; clear method works as expected.
- The iterator method returns a properly initialized set iterator, which can be used to retrieve elements of the set (in the order imposed by the underlying data structure).