# CSE 214 : Data Structures (Fall 2023) – Assignment I

Instructor: Dr. Ritwik Banerjee
Computer Science, Stony Brook University

This assignment consists of two parts, and 6 pages. Students are required to follow the development environment described for this course. Any code provided to you as part of this assignment can be found in the appendix sections of this document. **This assignment is due by 11:59 pm, Sep 22 (Friday)**.

Remember that whenever you are implementing a data structure corresponding to an abstract data type, you can write additional methods that are not required by the interface. These additional methods, however, are not a part of the data type's "contract", and therefore, must be private.

## I. Linked List

To start off, you must implement a usable `DoublyLinkedList` generic class. We use this name to avoid any confusion with the built-in class provided by the Java library. A fully functioning implementation requires three classes:

1. The `DoublyLinkedList` class, where each node maintains a `next` and a `previous` reference. This class must implement the `ListAbstractType` interface provided to you as part of this assignment (and **NOT** the built-in list interface provided in Java's library).
2. A `Node` class, which must be a private static class nested inside `DoublyLinkedList`.
3. A `DoublyLinkedListIterator` class, which must implement the `TwoWayListIterator` interface provided to you as part of this assignment.

1. **The `Node` class with its fields and constructor:** The very first step is to understand how to write the `Node` class as a private class nested inside `DoublyLinkedList`. Nested classes can be static or non-static, but in this assignment you will have to use a static nested class. The `Node` class will never be directly used by any external client code, so fields of this class can be public (remember, the entire class is nested and private, so these fields cannot be accessed externally even if they are individually declared to be anything other than private). To create a `DoublyLinkedList` with any elements, however, you will need to write a constructor for this `Node` class, and any `Node` instance must have a `next` and a `previous` reference along with its own element. (5)

2. **Implementing the `CollectionType` methods:** As we have discussed in the lectures, the data structure is a complete implementation of an abstract data type. In this assignment, your `DoublyLinkedList` data structure must implement the `ListAbstractType` interface. But this data type, in turn, extends the `CollectionType` interface. Thus, you must implement the following methods so that they function correctly as per the documentation provided in the `CollectionType` interface code: (15)

```
boolean add(E element);
boolean remove(E element);
int size();
boolean isEmpty();
boolean contains(E element);
```

3. **Implementing the `ListAbstractType` methods:** The `DoublyLinkedList` class will, of course, also need to implement the other additional methods declared in the `ListAbstractType` interface. These methods are more specific to the concept of a sequential collection of elements, and therefore, use of an element's "position" in a list. You must implement the following methods so that they function correctly as per the documentation in the `ListAbstractType` interface code: (15)

```
E get(int index);
E set(int index, E element);
void add(int index, E element);
void remove(int index);
```

4. **Implementing a two-way iterator:** In question 1, we did not include one very important method, which is inherited from the `java.lang.Iterable` interface. In your entire implementation of a doubly linked list, `Iterable` is the only interface readily available from a core Java library (unlike the other two built separately for this assignment: `CollectionType` and `ListAbstractType`). The `Iterable` interface requires implementating a method declared as (15)

    ```
    Iterator<T> iterator();
    ```

    But in our case, a special iterator is needed, in order to iterate forward *and* backward through a doubly linked list. So, Java's built in iterator (i.e., the object being returned by the above method signature) is not enough. So, you are provided with a sub-interface with additional methods, called the `TwoWayListIterator`. Your `DoublyLinkedList` class must therefore implement a method with the following signature:

    ```
    public TwoWayListIterator<E> iterator();
    ```

    The issue, of course, is that `TwoWayListIterator` is an interface, not a class! Which means there has to be a class that implements this interface. Implement a private class nested inside your linked list class, called `DoublyLinkedListIterator`, to accomplish this. Once again, since this is a nested class, carefully think about whether or not it should be static:

    - *Is an iterator associated with a specific list instance, or is an iterator something that is "shared" by the whole class?*

## II. Hot Potato!

There are $N$ people, numbered 1 to $N$, who are sitting in a circle. Starting at person 1, a hot potato is passed. After $M$ passes, the person holding the hot potato is eliminated, and the circle is reduced by 1 in size (i.e., there is no empty spot left after the elimination of a player). The game continues with the person who was sitting *after* the eliminated player picking up the hot potato. The last remaining player wins. So, if the game starts with $N = 5$ and $M = 0$, the players are eliminated immediately in the order $1, 2, 3, 4$, and player 5 wins. If the game starts with $N = 5$ and $M = 1$, then the players are eliminated in the order $2, 4, 1, 5$, and players 3 wins.

As you can probably guess, this game can be implemented using a linked list, where the hot potato being passed around can be thought as the "iteration" on that list. Your task is to write a program to play this game in two different ways:

5. Use your own `DoublyLinkedList` class to implement the circle of players. Thus, you will also need to use your own `DoublyLinkedListIterator` to simulate the passes, and your own implementation of methods such as `remove` to reflect the elimination of players. This must be implemented as the `playWithDoublyLinkedList(int, int)` method provided to you in `HotPotato.java`. You can add other methods to this class if you need. These additional methods must, however, be private. This exercise is meant to develop your insights about how to use data structures and data types you have built, and attempts to bypass that goal (by, say, internally using arrays or some other data structure readily available from Java) will be viewed as an incorrect solution to this question. (25)

6. The same as the previous question, but using Java's built in `java.util.LinkedList` class along with its iterator and methods. This implementation must be in the `playWithLinkedList(int, int)` method in `HotPotato.java`. This part of the assignment is included so that you get used to not just building your own data structures, but also gain experience in using built in data structures and data types. Here, too, the use of any built in data structures *other than linked list* will be viewed as an incorrect solution to this question. (25)

For both implementations, you can assume that the game will be played with at least two players (i.e., $N > 1$), and that the length of the pass will always be a non-negative value (i.e., $M \geq 0$).

## Really important notes

1. The correctness of your code will be tested with various test cases, and NOT by reading through your code. In lectures, we have discussed the role of a "client" code. For this assignment (and future assignments in this course), you should think of the grader's code as a client code. For example, the grader will test the `set(int index, E element)` method by creating a list using your `add(E element)` method a few times, and then calling the `set` method. The grader will not read your code line-by-line to assess its correctness!
2. Do not think of this assignment as an implementation of individual methods put together. Many methods are interrelated, and incorrect implementation of one method may affect the points obtained in other places. For example, if your `add(E element)` method does not work, the grader cannot even create a list with your code. In that case, the grader simply cannot grade any of the other methods either! Another (less drastic) example of such interdependence: the `size()` method can only be deemed correct if it reports the correct size of a list after several combinations of additions and removals. So, mistakes in the `add(E element)` and/or the `remove(E element)` methods can affect the points obtained for the `size()` method (e.g., add two items to an empty list, then remove one item, but a calls to `size()` still returns 2).
3. The `HotPotato` code also shows you how the main method will call your implementations of the game, and the expected outputs. The provided main method thus serves as a very basic client code, with two test cases. Students are expected to test their own code with as many test cases as they want.
4. A nice string representation of your linked list may be very useful for testing your own code, and printing out the lists that you obtain after various operations. To aid you with this, the `toString()` method is provided in the appendix. Please include this method *exactly as provided* in your `DoublyLinkedList` class. When grading, this method will be expected to be included, and it will be called to test the rest of your code for this class.
5. It might be helpful to remember that there is a very easy way to test if you have implemented the iterator: if you cannot use an *enhanced for loop* (also called the for-each loop) on your `DoublyLinkedList`, that means you have not implemented an iterator properly. A very simple client code for testing may be something like

```
public static void main(String... args) {
    DoublyLinkedList<Integer> nums = new DoublyLinkedList<>();
    System.out.println(nums.toString()); // should print "[]"
    nums.add(1);
    nums.add(2);
    System.out.println(nums.toString()); // should print "[1, 2]"
}
```

6. **What to submit?** You will notice in the appendix codes that you have to write all your code in a package called "cse214hw1". Once you have written and tested all your code, simply create a `.zip` of this package and submit it on Brightspace. Recall that a package is just a folder in your computer, so zipping the files works just like compressing and creating a .zip of any other folder. Do NOT submit your entire Java project!!
*Deviations from the expected submission format and/or the requirements mentioned earlier in this assignment carries varying degrees of score penalty (depending on the amount of deviation).*

## Appendix A: `CollectionType.java`

```
package cse214hw1;

/**
 * DO NOT MODIFY THIS CODE!!
 * @author Ritwik Banerjee
 */
```

```
public interface CollectionType<E> extends Iterable<E> {
    /**
     * Adds the specified element to this collection. If the collection offers a particular
     * iteration order, then the element is added at the end of the collection.
     *
     * @param element the specified element
     * @return {@code true} if the element was successfully added
     */
    boolean add(E element);

    /**
     * Removes the specified element (if it exists) from this collection. If the collection
     * offers a particular iteration order, then the first occurrence of this element (as
     * encountered in the iteration) is removed.
     *
     * @param element the specified element
     * @return {@code true} if the element was successfully removed from this collection
     */
    boolean remove(E element);

    /**
     * @return the number of elements in this collection
     */
    int size();

    /**
     * @return {@code true} if this collection contains no elements
     */
    boolean isEmpty();

    /**
     * Identifies whether or not this collection contains the specified element.
     *
     * @param element the specified element
     * @return {@code true} if the specified element was found in this collection
     */
    boolean contains(E element);
}
```

## Appendix B: `ListAbstractType.java`

```
package cse214hw1;

/**
 * DO NOT MODIFY THIS CODE!!
 * @author Ritwik Banerjee
 */
public interface ListAbstractType<E> extends CollectionType<E> {
    /**
     * @param index the specified position in this list
     * @return the element found at the specified position in this list
     */
    E get(int index);

    /**
     * Replaces the current element at the specified position with the provided new element.
     *
     * @param index   the specified position
     * @param element the new element provided
     * @return the element previously held at the specified position
     */
    E set(int index, E element);

    /**
     * Adds the specified element at the specified position, pushing subsequent elements to
     * one position higher. If {@code index == 0}, the addition happens at the front of this
     * list. If {@code index == size()}, the element is added as the new last element.
     *
     * @param index   the specified position
     * @param element the specified element to be added
     */
    void add(int index, E element);

    /**
     * Removes the element at the specified position.
     *
     * @param index the specified position
     */
    void remove(int index);
}
```

## Appendix C: `TwoWayListIterator.java`

```
package cse214hw1;

import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * DO NOT MODIFY THIS CODE!!
 * @author Ritwik Banerjee
 */
public interface TwoWayListIterator<E> extends Iterator<E> {
    /**
     * @return {@code true} if the reverse iteration has more elements. In other words,
     * returns {@code true} if {@link #previous} would return an element rather than
     * throwing an exception.
     */
    boolean hasPrevious();

    /**
     * Returns the previous element in this list and moves the cursor position backward
     * by one. This method may, if needed, be intermixed with calls to {@link #next()}
     * to go back and forth.
     *
     * @return the previous element in this list
     * @throws NoSuchElementException if the iteration has no previous elements
     */
    E previous();

    /**
     * Inserts the specified element into the list. The element is inserted immediately
     * before the element that would be returned by {@link #next()}, if any, and after
     * the element that would be returned by {@link #previous()}, if any. If the list
     * contains no elements, the new element becomes the sole element on the list. The
     * new element is inserted before the implicit cursor: a subsequent call to
     * {@code next} would be unaffected, and a subsequent call to {@code previous} would
     * return the new element.
     *
     * @param element the element to insert
     * @throws UnsupportedOperationException if the {@code add} method is not supported
     *                                       by this list iterator
     */
    void add(E element);


    /**
     * Replaces the last element returned by {@link #next()} or {@link #previous()} with
     * the specified element (optional operation).
     * This call can be made only if neither {@link #remove()} nor {@link #add(E)} have
     * been called after the last call to {@code next()} or {@code previous}.
     *
     * @param element the element with which to replace the last element returned by
     *                {@code next} or {@code previous}
     * @throws UnsupportedOperationException if the {@code set} operation is not
     *                                       supported by this list iterator
     * @throws IllegalStateException         if neither {@code next} nor {@code previous}
     *                                       have been called, or {@code remove} or
     *                                       {@code add} have been called after the last
     *                                       call to {@code next} or {@code previous}
     */
    void set(E element);
}
```

## Appendix D: Printing a `DoublyLinkedList`

```
@Override
public String toString() {
    Iterator<E> it = this.iterator();
    if (!it.hasNext())
        return "[]";
    StringBuilder builder = new StringBuilder("[");
    while (it.hasNext()) {
        E e = it.next();
        builder.append(e.toString());
        if (!it.hasNext())
            return builder.append("]").toString();
        builder.append(", ");
    }
    // code execution should never reach this line
    return null;
}
```

## Appendix E: `HotPotato.java`

```
package cse214hw1;

import java.util.LinkedList;

public class HotPotato {
    public static DoublyLinkedList<Integer> playWithDoublyLinkedList(int numberOfPlayers, int lengthOfPass) {
        return null; // TODO
    }

    public static LinkedList<Integer> playWithLinkedList(int numberOfPlayers, int lengthOfPass) {
        return null; // TODO
    }

    public static void main(String... args) {
        // in both methods, the list is the order in which the players are eliminated
        // the last player (i.e., the last element in the returned list) is the winner
        System.out.println(playWithDoublyLinkedList(5, 0)); // expected output: [1, 2, 3, 4, 5]
        System.out.println(playWithLinkedList(5, 1));       // expected output: [2, 4, 1, 5, 3]
    }
}
```