Data Structures	Assignment 1
CCO 2103-01, Spring 2024	Due at 5:00pm, Wednesday, April 3

Hand in your solutions electronically using LearnUs. As was announced in class, you are not allowed to use any data structure libraries, including the linked lists provided by JDK. When in doubt, contact the course staff.

This assignment is *not* a completion points assignment.

Submit your source code(s), zipped as `yourStudentID.zip`. For example, if your student ID is `2023000000`, then you must zip all your source code(s) into `2023000000.zip` and submit this file. Each class should have its own `.java` file, of which the filename is the same as the class name. Do *not* include your student ID as part of the class names.

This assignment consists of one programming task.

(1) (100 points) Write a class that implements a circular doubly linked list of integers with a header node, and also write an iterator for it. The integers can be positive, zero, or negative. In what follows, the *first node* of a list is defined as the following node of the header node; *last node* is the previous node of the header. Recall that the header node is not used to store a data item.

Your code must have the following three classes: `DNode`, `CDList`, and `CDIter`. The `DNode` class is the class you use to represent each node of the list, and must have the following fields:

- `public int data;`
  Holds the data of the node.
- `public DNode prev, next;`
  Holds the reference to the previous and the next node, respectively.

It is important to declare these fields (and what follows) as `public`, because our grading program will directly access these fields to check if they are correctly set.

`CDList` is the main class that implements the list. Note that multiple instances of `CDList` can be created. The class `CDList` must have the following fields and methods:

- `public final DNode header;`
  Holds the reference to the header node.
- `public CDList();`
  Constructs an empty list.
- `public void insertAtFront(int x);`
  Inserts a node with data `x` as the new first node of the list.
- `public void insertAtEnd(int x);`
  Inserts a node with data `x` as the new last node of the list.

- `public int deleteFromFront();`
  Returns `-1` if the list is empty. Otherwise, deletes the first node of the list and returns the data that was in the deleted node.

- `public int deleteFromEnd();`
  Returns `-1` if the list is empty. Otherwise, deletes the last node of the list and returns the data that was in the deleted node.

- `public CDIter getIter();`
  Returns an iterator that points to the first node of the list. If the list is empty, it returns an iterator that points to the end of the list, i.e., the header node.

Finally, the class `CDIter` is an iterator for `CDList`. Note that a multiple number of iterators can be created for a list. The class `CDIter` must have the following fields and methods:

- `public DNode cur;`
  Holds the reference to the current node.

- `public boolean atEnd();`
  Returns `true` if and only if the iterator is at the end of the list, i.e., pointing to the header node. Note that this is *different* from the case where the iterator is pointing to the last node of the list.

- `public int getValue();`
  Returns the data of the current node, without changing the position of the iterator. If the iterator is at the end of the list (i.e., the current node is the header node), returns `-1`.

- `public boolean setValue(int x);`
  Changes the data of the current node to `x` and returns `true`, without changing the position of the iterator. If the iterator is at the end of the list, does nothing and returns `false`.

- `public void prev();`
  Moves the iterator to the previous node.
  If the iterator was originally at the end of the list, it must point to the last node after the operation (unless the list is empty, in which case the iterator must remain at the header node).
  If the iterator was originally at the first node, it must point to the end of the list after the operation.

- `public void next();`
  Moves the iterator to the next node.
  If the iterator was originally at the end of the list, it must point to the first node after the operation (unless the list is empty, in which case the iterator must remain at the header node).
  If the iterator was originally at the last node, it must point to the end of the list after the operation.

- `public boolean delete();`
  Deletes the current node and returns `true`. After the deletion, the iterator must advance to the next node of the deleted node. If the iterator is at the end of the list, does nothing and returns `false`.

- `public void insertBefore(int x);`
  Inserts a node with data `x` before the current node. If the iterator is at the end of the list, the new node is to be inserted as the new last node of the list. The iterator should remain pointing to the same node before and after the operation.

- `public void insertAfter(int x);`
  Inserts a node with data `x` after the current node. If the iterator is at the end of the list, the new node is to be inserted as the new first node of the list. The iterator should remain pointing to the same node before and after the operation.

If there exists more than one iterator pointing to the same node, and the `delete()` method is called on one of them, the behavior of the other iterators is undefined. That is, you do not need to handle this case. However, it is allowed to call `delete()` method of an iterator if all other iterators of the list are pointing to other nodes. You can assume that `CDList.deleteFromFront()` and `CDList.deleteFromEnd()` will never be called when there exists an iterator pointing to the node to be deleted.

Every method (and the constructor `CDList()`) must run in $O(1)$ time. *You are free to add your own methods and fields as you see fit, but you need to implement all of those methods.* You do not need to provide an entry point to your code. Our grading program will use your class and check if your code works correctly. (Of course, you would need to write your own driver that uses your class to test your code before you submit your solution.)

(Extra question. Do NOT submit.) Make your iterator robust by allowing deletion of a node that is pointed to by multiple iterators. When a node is deleted, *all* iterators that were pointing to the node should advance to the next node. The execution time of the delete operation must be proportional to the number of iterators that are pointing to the node being deleted, not the number of all the iterators of the list.