CS 284: Homework 2
Due: Thursday, February 17th, 2022 at 11:59pm

1 Assignment Policies

Don't forget the honor pledge!

Collaboration Policy. Homework will be done individually: each student must hand in their own answers. It is acceptable for students to collaborate in understanding the material but not in solving the problems or programming. Use of the Internet is allowed, but should not include searching for existing solutions.

Under absolutely no circumstances code can be exchanged between students. Excerpts of code presented in class can be used.

Assignments from previous offerings of the course must not be re-used. Violations will be penalized appropriately.

2 Assignment

This assignment consists in implementing a double-linked list *with fast accessing*. Fast accessing is provided by an internal *index*. An index is just an array-based list that stores references to nodes. (Such a structure isn't actually useful in the world, but it's a nice way to simultaneously give you practice working with linked-list nodes *and* using the `java.util.ArrayList` interface.)

Before going further, let's take a step back and recall some basic notions regarding double-linked lists.

As explained in the lectures, a double-linked list (DLL) is a list in which each node has a reference to the next one and also a reference to the previous one. The corresponding Java class therefore has three data fields or attributes:

- `Node<E> head`
- `Node<E> tail`
- `int size`

You should, in general, access the elements of the list through the references `head` and `tail`. For example, the i-th element is obtained by starting from `head` and then jumping through $i-1$ nodes. You implementation includes an additional attribute, namely an *index*. An index is simply a list based array that stores the references to each node in the DLL. Since the access to an element in an array-based list is $\mathcal{O}(1)$. You should (a) maintain the index to be correct, and (b) give simpler implementations for methods involving indices.

As usual, please put everything in the `cs284` package.

2.1 Design of the Class `IDLList<E>`

2.1.1 The Inner Class `Node<E>`

First of all, an inner class `Node<E>` should be declared. This class should include three data fields:

- `E data`
- `Node<E> next`
- `Node<E> prev`

It should also include the following operations:

- `Node(E elem)`, a constructor that creates a node holding `elem`.
- `Node(E elem, Node<E> prev, Node<E> next)`, a constructor that creates a node holding `elem`, with `next` as next and `prev` as prev.

2.1.2 The Class `IDLList<E>`

The class `IDLList<E>` should include the declaration of this inner private class `Node<E>`. Apart from that, it should have four data fields:

- `Node<E> head`
- `Node<E> tail`
- `int size`
- `ArrayList<Node<E>> indices`

Note that `indices` is an array-based list of references to nodes. A reference to the first element of list is therefore available as the first element of `indices`. A reference to the second element of the list is therefore the second element in `indices`. And so on.

You are requested to implement the following operations (a summary is provided at the end of this assignment, in a UML diagram) for `IDLList<E>`:

- `public IDLList()`, that creates an empty double-linked list.

- `public boolean add(int index, E elem)` that adds `elem` at position `index` (counting from wherever head is). It uses the index for fast access. Following `java.util.List`, valid indices are in the range 0 to size (inclusive). (Calling `add(size, elem)` makes `elem` the new tail.) It always returns true.
- `public boolean add(E elem)` that adds `elem` at the head (i.e. it becomes the first element of the list). It always returns true.
- `public boolean append(E elem)` that adds `elem` as the new last element of the list (i.e. at the tail). It always returns true.
- `public E get(int index)` that returns the object at position `index` from the head. It uses the index for fast access. Indexing starts from 0, thus `get(0)` returns the head element of the list.
- `public E getFirst()` that returns the object at the head.
- `public E getLast()` that returns the object at the tail.
- `public int size()` that returns the list size.
- `public E removeFirst()` that removes and returns the element at the head. Should throw an `IndexOutOfBoundsException` if there is no such element.
- `public E removeLast()` that removes and returns the element at the tail. Should throw an `IndexOutOfBoundsException` if there is no such element.
- `public E removeAt(int index)` that removes and returns the element at the index `index`. Use the index for fast access. Should throw an `IndexOutOfBoundsException` if there is no such element.
- `public boolean remove(E elem)` that removes the first occurrence of `elem` in the list and returns true. Return false if `elem` was not in the list.
- `public String toString()`. That presents a string representation of the list. It should use a Python-like notation for output, with square brackets and commas, deferring to `E`'s `toString()` method for the elements. For example, if your list `l` contains the strings `"hello"` and `"how are ya"` and `"goodbye"`, in that order, then `l.toString()` should yield `"[hello, how are ya, goodbye]"`.

The following operations require index maintenance (i.e. they have to assign or modify the index):

- `public IDLList()`.
- `public boolean add(int index, E elem)`.
- `public boolean add(E elem)`.
- `public boolean append(E elem)`.
- `public E remove()`.

- `public E removeLast()`.
- `public E removeAt(int index)`.
- `public boolean remove(E elem)`.

2.1.3 The Class `IDLListTest`

We expect you to write tests for your code; they are part of your submission. You should use JUnit.

We will grade your tests based on how well they rule out bad implementations while accepting good ones. There are two key points here that you should keep in mind while writing your tests:

- Your tests should not be too restrictive. *Any* correct implementation should pass your tests, not just *your* implementation.
- Your tests should not be too permissive. Incorrect implementations should *not* pass your tests.

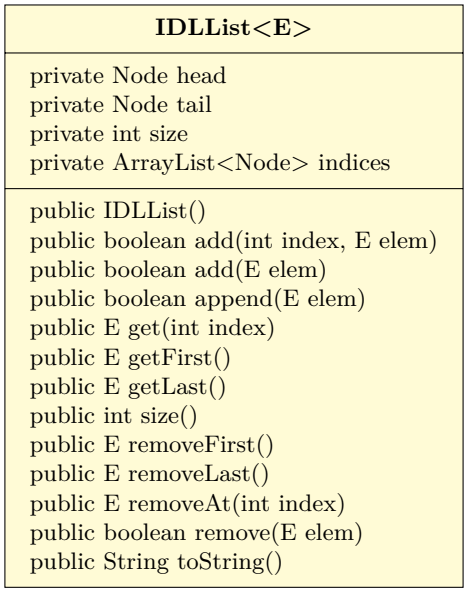
Think carefully about what behaviors to test for!

3 Submission instructions

Submit a single file named `IDLList.zip` through Canvas that includes `src/cs284/IDLList.java` and `src/cs284/IDLListTest.java` with your test cases. No report is required. Your grade will be determined as follows:

- You will get 0 if your code does not compile.
- The code must implement the following UML diagram precisely.
- We will try to feed erroneous and inconsistent inputs to all methods. All arguments should be checked. Throw an `IllegalArgumentException` for bad arguments in general—but you should use a `IndexOutOfBoundsException` for index-related errors.
- Partial credit may be given for style, comments and readability.

The class `IDLList<E>` should include the following operations:



The private inner class `Node` should follow the UML diagram. Note that since it's an *inner* class, it inherits the parent class `IDLList<E>`'s parameter `E`, and doesn't get parameterized itself: