CS 284D: Homework Assignment 2
Due: Feburary 22nd, 11:55pm

1 Assignment Policies

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. 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`

Accessing the elements of the list is therefore realized 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. Indeed, just like single-linked lists, accessing an element in a DLL is of time complexity $\mathcal{O}(n)$. In order to alleviate this situation this assignment asks you to implement an enhanced DLL, *Indexed DLL* or IDLL. An IDLL 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 $\mathcal{O}(1)$, this will allow the users of IDLL to enjoy the benefits of fast access, and at the same time, use a list implementation which does not waste memory given that it may shrink or grow dynamically, a property which is known to be one of the advantages of linked-lists in general.

The way faster access is achieved is that the `get(int i)` operation, in its implementation, rather than starting from the head of the list and traversing each node until the i-th node is reached, it simply uses the `get(int i)` operation of an array-based list or index called `indices` which it maintains, together with the other data fields.

This does come at a price though. We need more memory to store the array-based list `indices` for one thing. Another is that all the operations of IDLL will have to maintain the indices up to date. For example, whenever a new element is added to the DLL, the array-based indices will have to be updated by inserting the new reference.

You are requested to implement a class `IDLList<E>` that encodes Indexed DLLs, following the guidelines presented in the next section.

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 IDList ()`, 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. 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 getHead ()` 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 remove()` that removes and returns the element at the head. Should throw an `IllegalStateException` if there is no such element.

- `public E removeLast ()` that removes and returns the element at the tail. Should throw an `IllegalStateException` 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 `IllegalStateException` 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.

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).`

3   Submission instructions

Submit a single file named `IDLList.zip` through Canvas that includes `IDLList.java` and `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.

- Partial credit may be given for style, comments and readability.

The private inner class `Node<E>` should follow the UML diagram:

Node[E]
E data   Node[E] next   Node[E] prev
Node (E elem)   Node (E elem, Node[E] prev, Node[E] next)

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

4

IDLList[E]
private Node[E] head private Node[E] tail private int size private ArrayList[Node[E]] indices
public IDLList () public boolean add (int index, E elem) public boolean add (E elem) public boolean append (E elem) public E get (int index) public E getHead () public E getLast () public int size() public E remove () public E removeLast () public E removeAt (int index) public boolean remove (E elem) public String toString()