Data Structures and Algorithms **Spring '23 section 1** **Dr. Hristescu**

Lab 4

Assigned: Thursday, February 9 **Due: 10:00pm** Monday, February 13

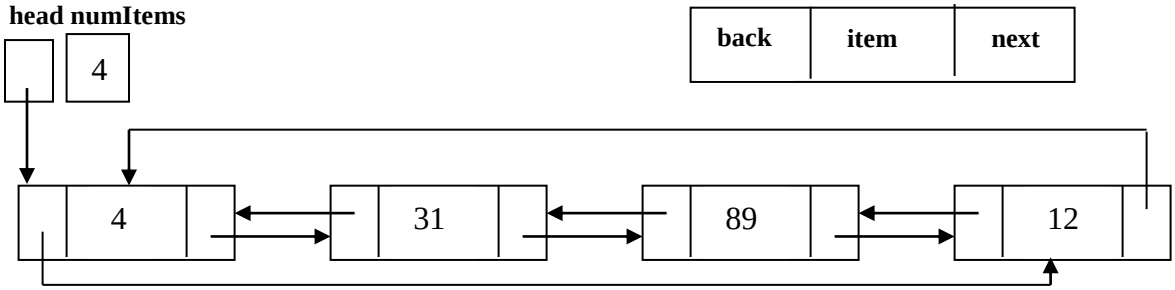
For this lab we will use a ***circular doubly linked structure (CDLS)*** with a **head** reference to implement the List ADT. The goal for the doubly linked structure is, as discussed in class, to increase the efficiency of the implementation by reducing the number of links traversed to ***at most*** **(in the worst case)** half the links in the whole linked structure. We will use just one entry point in this linked structure (in this case a **head** reference) in order to eliminate redundancy and reduce the number of data fields that have to be maintained.

When a ***circular doubly linked structure (CDLS)*** is used to implement a **List** ADT (see below):

- The node at the back of the list, instead of having a **null** *next* link, references the node at the front.
- The node at the front of the list, instead of having a **null** *back* link, references the node at the back.
- Since there needs to be an entry point in the CDLS, that entry is a **head** reference for this lab.
- If the list is empty head is null, otherwise **head.getBack()** references the node at the back of the list(≡tail).

Note: No link in the circular doubly linked structure **(next or back)** can ever be **null!!!**

CDLS structure: **head** and **numItems** data fields.



DNode structure**:**

Note: The item data field in the DNode class is a reference data field, not a primitive type (int), the way it is depicted only for simplicity in this picture.

Problem 1: Implement the ListInterface as well as a **toString** and **toStringR** **(efficiently collects the items from last to first using the underlying data structure- both methods should NOT have any calls to get or find but traverse the data structure and collect the items in the correct order)** method using a circular doubly linked structure in class **ListCDLSBased** and test the functionality of your class using a menu-driven test application with the following options:

0. Exit program
1. Insert item into the list
2. Remove item from the list
3. Get item from the list
4. Clear the list
5. Display size and content of the list in order and in reverse order
6. Delete the smallest and largest item in the list
7. Reverse the list

Your submission should include **at least** a sample run of your program on this input data that should look like this.

Tackle your design in 4 steps performed in this order:

Step1: Design the encapsulated node structure (**DNode**) to include an additional data field called **back** of type DNode with the necessary functionality. Design the 1-parameter constructor with **self-referencing** links for **next** and **back**.

Step2: Make sure you have the **correct toString** method from Lab3 (traverses the doubly linked structure and collects all items from first to last) – should work with just a simple change: the method collects numItems items by traversing the necessary number of links rather than testing for the end of the linked structure by checking if the reference is null – it never will be for a circular structure. Design the **toStringR** method that collects the items (using the using back links in the doubly linked structure) from last to first. Neither toString nor toStringR can be implemented to make any calls to the get or find method.

Step3: Re-design the find method to handle an index in an extended range [0, listsize]. The redesigned method should efficiently make use of the circular doubly linked structure. Using the old find from ListReferenceBased would be both incorrect and **defeat the purpose** of this lab.

Step4: Design add and remove to use it efficiently (these methods will not work with the old find, as it cannot handle indices beyond [0, listsize-1]). **Use the design guidelines learned in class**: develop the code for the general case, check to see if/what boundary cases need to be treated separately and incorporate them before the code for the general case. Make sure that both position-related boundary cases as well as collection-size boundary cases are treated and that both **head** and **numItems** are updated when affected (d.f. are consistent with the state of the collection at all times and that adding to an empty list and removing from a 1-item list correctly updates both these data fields).

Note:

- Build DNode on the Node class frame and re-use driver and methods developed for Lab 3 instead of starting from scratch. Make sure option 5 calls **both** toString and toStringR and then displays the returned string.
- Make sure your implementation doesn't leave the CDLS in an inconsistent state at any point.

ExtraCredit I: Implement the ListInterfacePlus using an inner class that implements ***Iterator***. To test it, add an option 8 that returns an Iterator and offers a submenu that allows retrieving the current item or advancing through the collection.

8. Get an Iterator.

ExtraCredit II: implement and test a ***ListIterator*** instead of Iterator.

Problem 2: CDLS analysis for a list of **n** items:

a) **Space Complexity** analysis: Analyze the amount of memory used by the CDLS (**exact** amount of 4 bytes)

b) **Time Complexity** analysis: Analyze the efficiency of this implementation (include *best*, *worst* **and** *average* case analysis for methods **get**, **add** and **remove**).

Problem3: Lab4Conclusions: Summarize in a few sentences what you have learned by working on this lab.

Submit Follow the instructions to submit the lab electronically using **Lab4** for the name of your directory.

- P1: code for **DNode**, **ListCDLS** and **Driver** as well as sample runs (including on provided input data)
- P2: The Space and Time Complexity analysis of the CDLS implementation
- P3: Status and Conclusions