CSC 205

Fundamentals of Computer Science II

Spring 2017

<u>Project 1</u>

Due by class on Monday, March 27

Linked Lists are a fundamental data structure in computing. In class we studied the doubly-linked, circular list which has a dummy header node. But there are many variations on the concept of a Linked List. A Linked List can be:

- doubly linked or singly linked
- circular or non-circular
- with or without a dummy header node
- with or without a “finger” (or iterator) that points to the most recently processed node in the list

For this project you will implement one of these forms of a Linked List. Since Linked Lists are pointer-based structures, it can be difficult to visualize the structure when writing code. Therefore, we will be using the software of the BRIDGES project, which provides a mechanism to display your Linked List using nodes and arrows.

Earlier in the semester we implemented the classic BubbleSort algorithm. BubbleSort sorts a list of data by sweeping through the list, from left to right, comparing each pair of <u>adjacent</u> elements, and if those elements are out-of-order, then they are swapped in the list. BubbleSort repeats this sweep down the list until no swaps are done (at which time the list is sorted). The sweep down the list can be repeated at most n times (where n is the number of items in the list). Therefore the total number of swaps performed is at most $O(n^2)$. We also saw that the swap of two data items can be performed using either .set() and .get() methods, or using .add() and .remove() methods. In an ArrayList, the .set() and .get() take only constant time (i.e., $O(1)$), so the worst-case running time of BubbleSort is $O(n^2)$. In contrast, using .add() or .remove() requires “shoving over” a large number of data items. So the running time of BubbleSort when we use .add() and .remove() to swap is $O(n^3)$. This is much, much slower than $O(n^2)$.

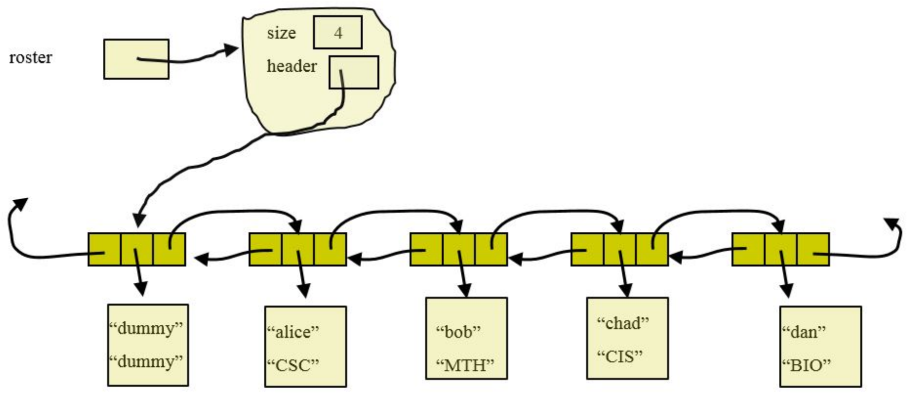
For this assignment you must complete the body of the given class CSC205_Project_1_Linked_List. This list will the following properties:

- doubly linked
- non-circular
- with a dummy header node
- with a “finger” (or iterator) that references the most recently processed node in the list

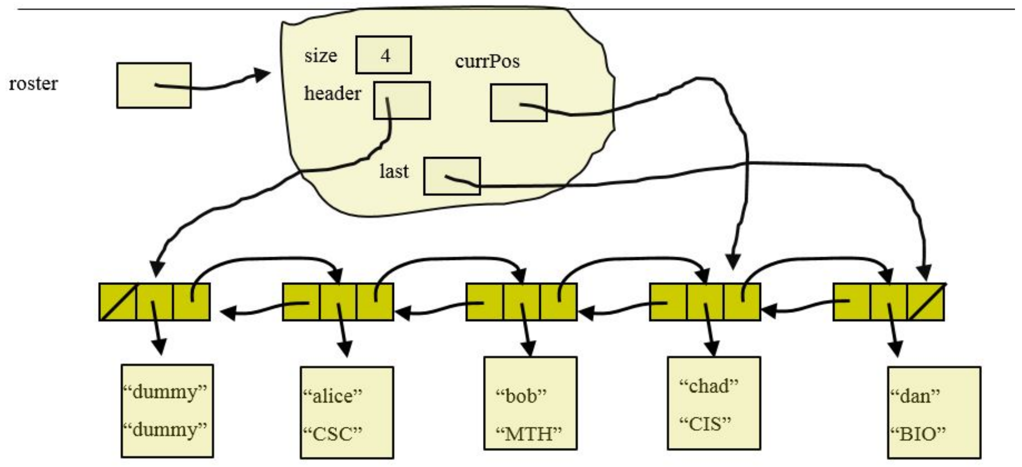
This assignment is similar to Lab 04. In that lab you needed to write 4 methods for the LucasLinkedList class, which was:

- doubly linked
- circular
- with a dummy header node
- without a "finger" (or iterator) that references the most recently processed node in the list

For example, a LucasLinkedList containing 4 items looks like:



In contrast, a CSC205_Project_1_Linked_List (that you must implement for this project) containing 4 items looks like:



The purpose of the currPos field (aka the "finger") is so that the List always has a pointer to the **most recently accessed** node in the list. For example, if your List performs a .get(45), then after locating the data in position 45 in the list, the currPos pointer should be pointing to the Node at position 45. If the next operation is .get(47), then your List should be able to locate position 47 by taking only 2 steps down the list from the current position. Naturally, after the .get(47) operation, the currPos pointer should now be pointing to the 47th Node in the List. All list operations should be implemented to take advantage of the currPos pointer. If the target location is closer to currPos than to the start or end of the list, then your code should move through the list from currPos to locate the target list entry.

Note that we have placed a “dummy” Student object into the dummy-header-node, rather than leaving this data field simply null (as was done in Lab 04). It should not matter what is contained in the data field at the dummy-header-node, since your code should never try to read that entry (why should it? It does not have any useful information). But to insure that your code interacts properly with the BRIDGES software, you should place a “dummy” Student object in the dummy-header-node.

A Linked List must use a Node object (as a container to hold the **next**, **prev** and **data** fields). In the LucasLinkedList class that you worked with in Lab04, the Node class was defined **inside** the LucasLinkedList class. In this project you will use the DLelement class from the BRIDGES software (just as you did in Lab05). This will allow you to visualize your CSC205_Project_1_Linked_List.

You must complete the following methods inside the given CSC205_Project_1_Linked_List.

- public E set (int i, E newItem)
- public E get (int i)
- public E add (E newItem)
- public E add (int i, E newItem)
- public E remove (int i)
- public DLemement<E> getDummy()
- public DLemement<E> getFinger()

The getDummy() and getFinger() methods should return a pointer/reference to the DLelement (aka Node) that is the dummy-header-node and the most-recently-accessed-node, respectively. Of course, you can create as many private (aka “helper”) methods inside the CSC205_Project_1_Linked_List that you wish.

After you have completed the implementation of the CSC205_Project_1_Linked_List, you will use that data structure to run the given BubbleSort algorithm, which is provided in the given Driver.java class. You should first run BubbleSort using the small “hard-wired” data size (which has n=5). The Driver will visualize the list after each swap, highlighting the “current position” node in green. You should be able to verify that the list is indeed sorted.

The Driver.java will display a GUI (graphical user interface) that prompts you to enter the parameters of your experiment (how many items should be sorted, etc.). The Driver uses code from the JavaFX library. This library is not automatically available to your code; you must explicitly set-up your project as follows:

1) Right click on your Project in Eclipse, select **Build Path** and then click on **Configure Build Path ...**
2) In the window that pops up, click on the **Libraries** tab.
3) Click on **Add External Jars ...**

4) In the window that pops up, navigate to the javafx file in

C:\Program Files(x86)\Java\jre1.8.0_92\lib\ext\jfxrt and click **Open**

5) Click **OK**.

You should then execute BubbleSort on larger sized input data. Execute the BubbleSort using at least 5 different values of n. I suggest using 1000, 2000, 3000, 4000 and 5000. But if these values ever take more than 15 minutes each to run, then try using smaller values of n.

You should run BubbleSort for each value of n for each type of Linked List, i.e., Java's built-in LinkedList and your CSC205_Project_1_Linked_List. Execute BubbleSort using both the set/get version of the code, and also the add/remove version of the code. Record the runtime in each case. You must turn in a report which includes the runtime data you collected (this will be 20 data points).

Your project report should also include an analysis of the worst-case big-Oh runtime of BubbleSort for <u>each</u> of the 4 versions of the algorithm:

1. Using Java's built-in Linked List, using the .set and .get methods
2. Using Java's built-in Linked List, using the .add and .remove methods
3. Using the CSC205_Project_1_Linked_List, using the .set and .get methods
4. Using the CSC205_Project_1_Linked_List, using the .add and .remove methods

For each of these 4 cases, what is the big-Oh worst-case run time of BubbleSort? Is it O(n), or O(n^2), or O(n^3) or something else? Explain and justify your answer. Your report should include at least one full page of prose, single-spaced, as well as the runtime data results.

<u>Deliverables:</u>

- Complete the Bridges survey at https://unccpsych.az1.qualtrics.com/jfe/form/SV_dcK4tmEmIQ68hDv and take a screen shot of the survey after you have completed it.

- Upload a **<u>zipped</u>** file of the CSC205_Project_1_Linked_List.java file, your survey screenshot, and a Word or pdf document containing your report.

- Hand in a paper copy of your report and your CSC205_Project_1_Linked_List.java file. Be sure that your code is readable (the code must not spill over the line onto the next line). Your code must use good programming style (proper indentation, good use of comments, well chosen, meaningful variable names, etc.)

If you have difficulty with the BRIDGES software, or seeing your visualization, then you can contact tues-bridges-group@uncc.edu (use the Subject line: **Bridges Query**).