Homework 3 :: Linked List

15-121 Fall 2015
Due: September 22, 2015 (11:50pm)
Basecode, Hand-In

Overview

This homework is comprised of two parts: you will be implementing your own `SinglyLinkedList` class and then a `DoublyLinkedList<E>` class. For `SinglyLinkedList`, you must use the `SingleNode` class provided to you or you will not receive credit, and everything in that part of the homework must be iterative — no recursion. For `DoublyLinkedList<e>`, you must use the `DoubleNode<E>` class.

As with all assignments, you will be graded in part on your coding style. Your code should be easy to read, organized, and well-documented. Be consistent in your use of indentation and braces. See the style guide for more details.. This homework will be graded with full style penalties (not half, like homeworks 1 and 2).

Background, part 1 :: SinglyLinkedList

For the first part of this homework, you will be creating your own class, `SinglyLinkedList<E>`, and you will do so using the `SingleNode` class. Examine the `SingleNode` class carefully. To create a new `Node`, you would do the following:

```
SingleNode <name> = new SingleNode(<>);
```

The data in a `SingleNode` is always a String, and the next pointer in these nodes is by default set to null. Once you've created it, you can change the `next` and `data` variables as follows:

```
SingleNode node = new SingleNode("foo");
node.next = new SingleNode("bar"); // adds another node after "foo"
node.data = "baz"; // change the data of the first node from "foo" to "baz"
```

Make sure you fully understand the `SingleNode` class before proceeding. You will be using it heavily throughout this exercise.

Next, examine the `SinglyLinkedList` class carefully. Notice there is a global variable named `numElements` and another global variable named `head`. The importance of these variables was discussed in lecture. Please refer to your notes if you are unsure of how you should use them.

Exercise :: Identify

Download the Basecode and fill in the required fields like so:

```
/**
```

```
 * @author [First Name] [Last Name] <[Andrew ID]>
 * @section [Section Letter]
 */

/**
 * @author Jess Virdo <jvirdo>
 * @section A
 */
```

Exercise :: concatenate

Complete the concatenate method which combines the `data` attributes from all `SingleNode`s in the `SinglyLinkedList`. Beginning with the `head`, combine all the data together. For example:

```
SinglyLinkedList list1 = new SinglyLinkedList();
// Items are inserted at the front.
list1.add("foo");
list1.add("bar");
list1.add("baz");
String ret = list1.concatenate(); // ret = "bazbarfoo"

SinglyLinkedList list2 = new SinglyLinkedList();
list2.concatenate(); // returns ""
```

Note: The quotation marks around the "bazbarfoo" method above indicate that the return type is a String. You should not add the quotation marks in your answer. This is just syntactically how we represent a String in Java on "paper". Also notice that there is no "separator" or space between the node values. They are simply meshed together in the order in which they appear. You will not receive credit for this method if it does not behave exactly as described above.

Exercise :: insertAfter

Complete the `insertAfter` method which searches through the `SingleNode`s in the `SinglyLinkedList` for the `findData` parameter. At the first occurrence of `findData`, it should create a new `SingleNode` with the given new data and insert that directly after the node with `findData`. If there are multiple occurrences of `findData`, only the closest to the front of the `SinglyLinkedList` should be used. If there are no occurrences of `findData`, the `SinglyLinkedList` should remain unchanged.

```
SinglyLinkedList list = new SinglyLinkedList();
list.add("foo");
list.add("bar");
list.add("zip");
// [HEAD] => zip => bar => foo => [TAIL]
list.insertAfter("cookie", "bar");
// [HEAD] => zip => bar => cookie => foo => [TAIL]

SinglyLinkedList list = new SinglyLinkedList();
list.add("foo");
list.add("bar");
list.add("zip");
// [HEAD] => zip => bar => foo => [TAIL]
list.insertAfter("cookie", "zippo");
// [HEAD] => zip => bar => foo => [TAIL]
```

Exercise :: buildList

Complete the `buildList` method which constructs a `SinglyLinkedList` from the given `ArrayList` parameter. The `SinglyLinkedList`'s data must be in the same order as the original `ArrayList`. This method should do nothing if `ArrayList` is empty. You must use the `add` method defined for you.

```
ArrayList<String> arr = new ArrayList<String>();
arr.add("zip");
arr.add("bar");
arr.add("foo");

SinglyLinkedList list = new SinglyLinkedList();
list.buildList(arr);
// list: [HEAD] => zip => bar => foo => [TAIL]

ArrayList<String> arr = new ArrayList<String>();
SinglyLinkedList list = new SinglyLinkedList();
list.buildList(arr);
// list: [HEAD] => null => [TAIL]
```

Exercise :: equals

Complete the `equals` method. Two `LinkedLists` are considered "equal" if they contain all the same `SingleNodes` (i.e. not literally the some node objects, but nodes with the same data), in the same order. The following is the algorithm you should use to implement this method. If you use a different algorithm, you will not receive credit.

1. Compare the sizes of the 2 lists using the `size` method provided for you. If two lists are not the same size, there is no way they can be equal. Since this is an `O(1)` runtime, we can mitigate unnecessarily looping through each `SingleNode` quickly.
2. If the lists are the same length, we can check each individual `SingleNode` and compare it against the `SingleNode` in the same relative position in the other list (self quiz — why don't we need to worry about `NullPointerExceptions`?).
3. If any 2 `Nodes` differ, this method should `return false`.
4. Once you have checked every `Node`, this method should `return true`.

```
SinglyLinkedList list1 = new SinglyLinkedList();
list1.add("foo");
list1.add("zip");

SinglyLinkedList list2 = new SinglyLinkedList();
list2.add("foo");
list2.add("zip");

boolean ret = list1.equals(list2); // ret = true
```

Exercise :: bringToFront

Complete the `bringToFront` method which takes an `index` as a parameter. The `SingleNode` at that `index` is moved to the front (i.e. made the first item) in the `SinglyLinkedList`. The rest of the `SingleNodes` should remain in their same relative positions.

If the given `index` is larger or equal than the number of `Nodes` in the `SinglyLinkedList`, this method should do nothing. You may assume that `index` will be greater than or equal to zero. Remember that in Computer Science, we begin counting from 0 (not 1).

```
SinglyLinkedList list = new SinglyLinkedList();
list.add("foo");
list.add("bar");
list.add("zip");
// [HEAD] => zip => bar => foo => [TAIL]
list.bringToFront(2);
// [HEAD] => foo => zip => bar => [TAIL]

SinglyLinkedList list = new SinglyLinkedList();
list.add("foo");
list.add("bar");
list.add("zip");
// [HEAD] => zip => bar => foo => [TAIL]
list.bringToFront(3);
// [HEAD] => zip => bar => foo => [TAIL]
```

Exercise :: removeAll

Complete the `removeAll` method which removes all `SingleNodes` from the `SinglyLinkedList` whose `data` exactly match the given `length` parameter. If more than 1 occurrence of a `SingleNode`'s String of length `length` exist in the `SinglyLinkedList`, all occurrences of should be removed. If there are no strings of length `length`, this method should not change the list.

```
SinglyLinkedList list = new SinglyLinkedList();
list.add("foo");
list.add("zip");
list.add("fish");
// [HEAD] => fish => zip => foo => [TAIL]
list.removeAll(3)
// [HEAD] => fish => [TAIL]

SinglyLinkedList list = new SinglyLinkedList();
list.add("foo");
list.add("zip");
list.add("fish");
// [HEAD] => fish => zip => foo => [TAIL]
list.removeAll(5)
// [HEAD] => fish => zip => foo => [TAIL]
```

Exercise :: reverse

Complete the `reverse` method which reverses the order of all the `Nodes` in the `LinkedList` in place. You may not create another `LinkedList` or any other data structure.

```
SinglyLinkedList list = new SinglyLinkedList();
list.add("foo");
list.add("bar");
list.add("zip");
// [HEAD] => zip => bar => foo => [TAIL]
list.reverse();
// [HEAD] => foo => bar => zip => [TAIL]
```

```
LinkedList list = new LinkedList();
// [HEAD] => null => [TAIL]
list.reverse();
// [HEAD] => null => [TAIL]
```

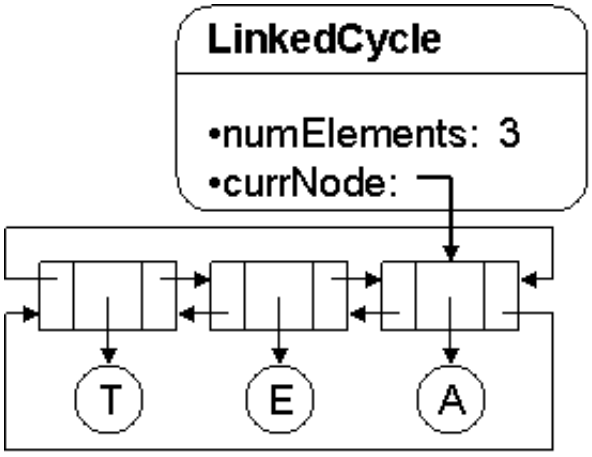
Background, part 2 :: DoublyLinkedList

A DoublyLinkedList<E> is a circular, doubly-linked list which uses DoubleNode<E>s. You should carefully read the DoubleNode class; it's different from the SingleNode class you were using in the first part of this homework. Another important difference between the first and second parts of this homework is that DoublyLinkedLists are parameterized (like ArrayLists), but SinglyLinkedLists were not.

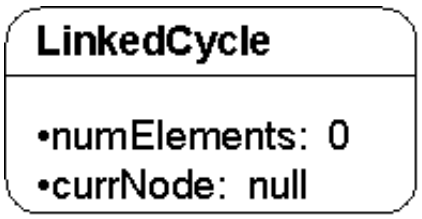
Consider the following code segment, which illustrates how the DoublyLinkedList class behaves:

```
DoublyLinkedList cycle = new DoublyLinkedList();
cycle.add("E");
cycle.add("A");
cycle.add("T");

int n = cycle.size();   // returns 3

String s = cycle.get(); // returns T
cycle.scroll(1);        // scrolls forward 1 position (wrapping around)
s = cycle.get();        // returns E

cycle.scroll(-5);       // scrolls backward 5 positions (wrapping around twice)
s = cycle.get();        // returns A
```

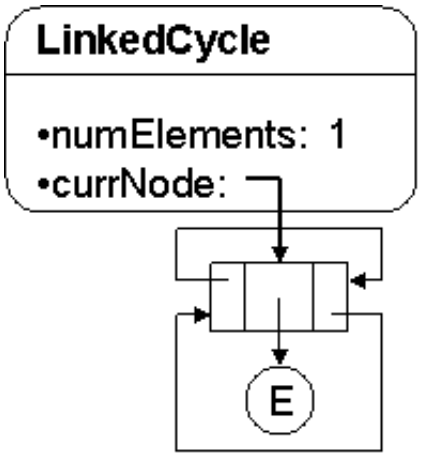
A DoublyLinkedList may contain duplicate values. The DoublyLinkedList data values are stored in a circular doubly-linked list. The following diagram depicts the DoublyLinkedList after the code segment shown earlier has executed:



The numElements field keeps track of the total number of elements that appear in the DoublyLinkedList, and the currNode field refers to the DoubleNode that stores the "current" value (the one that would be returned by the get method). Each DoubleNode has a data value (of any type), a next node, and a previous node. These nodes form a circular doubly-linked list. In these diagrams, the right side of each node contains the reference to the next node, and the left side of each node contains the reference to the previous node. An empty DoublyLinkedList is represented as follows.



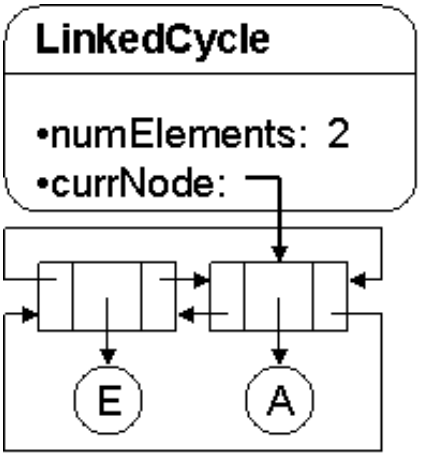
A `DoublyLinkedList` with only one element is represented as follows:



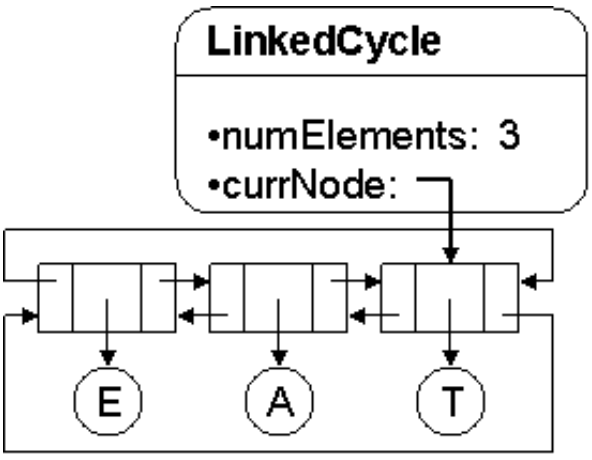
Note: Both `DoublyLinkedList` and `DoubleNode` are generic types. We will always refer to them using the syntax `DoublyLinkedList<SomeType>` and `Node<SomeType>`.

Here are a list of things you should complete, in order, to implement the `DoublyLinkedList` class.

1. Complete the `size` method, which returns the number of data values in the `DoublyLinkedList` in O(1) time.
2. Complete the `get` method, which returns the current data value in O(1) time. If the `DoublyLinkedList` is empty, this method should throw a `NoSuchElementException`.
3. Complete the `add` method, which adds the given element after the current data value (if any). After this method executes, `currNode` should refer to the node with the new element, and `currNode.prev` should refer to the node that used to be `currNode` (if any). For example, if the DoublyLinkedList initially appears as follows:

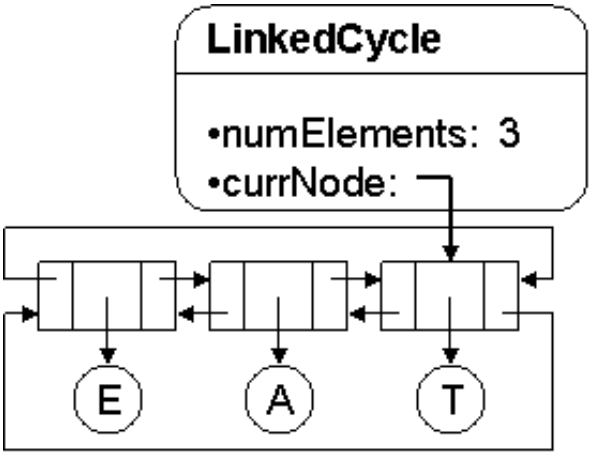
   

   then, after calling `add("T")`, the `DoublyLinkedList` should now appear as follows:



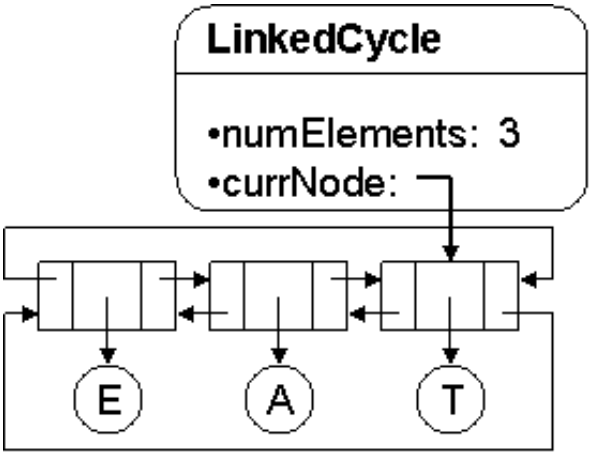
This method must run in O(1) time!

4. Complete the `toList` method, which should return a `List` (an instance of a class that implements Java's `List` interface) containing all the data values in the `DoublyLinkedList`, ending with the current data value. For example, if the `DoublyLinkedList` appears as follows:

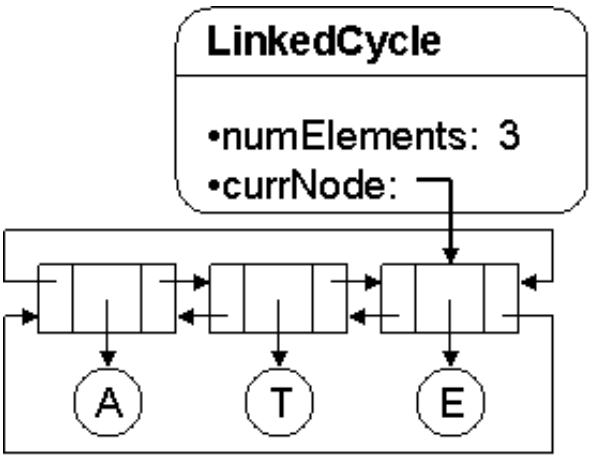


Then, `toList` should return the list `[E, A, T]`. This method should not remove any elements from the `DoublyLinkedList`. It must run in O(n) time.

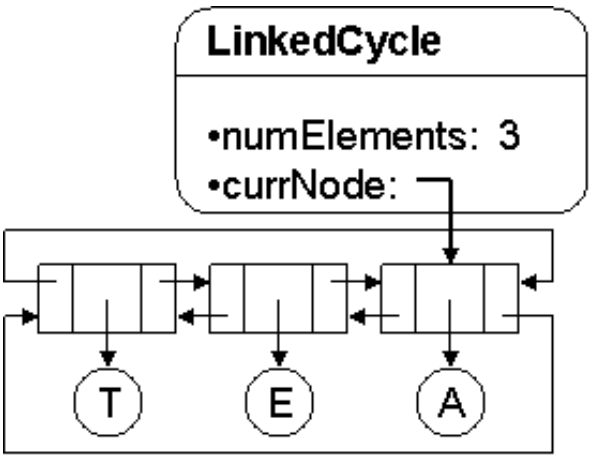
5. Complete the `contains` method, which returns `true` if the given element appears in the `DoublyLinkedList`, and returns `false` otherwise. This method must run in worst-case O(n) time and best-case O(1) time.
6. Complete the `scroll` method, which should advance `currNode` forward (following "next" references) by the given number of nodes. If the given number is negative, then scroll will scroll backward (following "prev" references) in the cycle. For example, if the `DoublyLinkedList` appears as follows:



After calling `scroll(1)`, the `DoublyLinkedList` will now appear as follows:

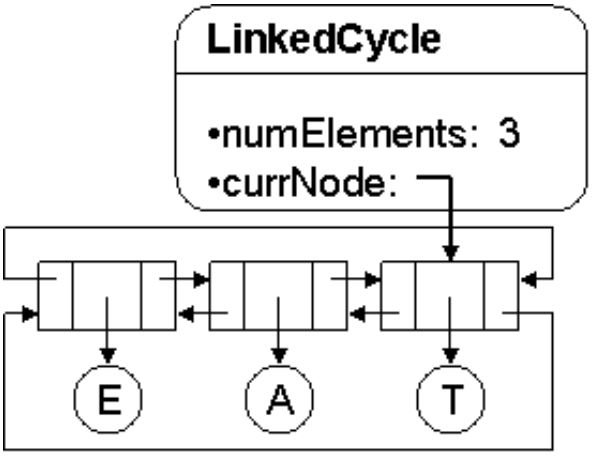


If we then call `scroll(-5)`, the `DoublyLinkedList` will now appear as follows:

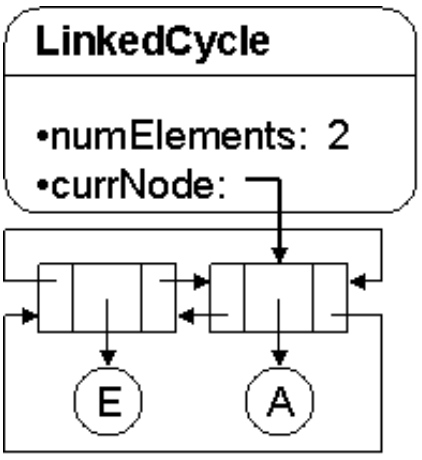


If the `DoublyLinkedList` is empty, this method should throw a `NoSuchElementException`. Otherwise, this method must run in O(n) time, where n is the absolute value of the argument value.

7. Complete the `remove` method, which removes and returns the current data value (if any). After the method has executed, `currNode` should refer to the node before the removed one (if any). Thus, if any nodes remain after remove has executed, `currNode.next` will refer to the node that used to be `currNode`. For example, if the `DoublyLinkedList` initially appears as follows:



After calling remove, the `DoublyLinkedList` should now appear as follows:



If the `DoublyLinkedList` is empty (and therefore there is no element to remove), then remove should throw a `NoSuchElementException`. This method must run in O(1) time.

Submitting your Work

When you have completed the assignment and tested your code thoroughly, create a `.zip` file on your work. Only include the following file(s):

1. `SinglyLinkedList.java`
2. `DoublyLinkedList.java`

Do not include any `.jar` or `.class` files when you submit, and zip only the files listed above. Do not zip not an entire folder containing the file(s).

Once you've zipped your files, visit the Autolab site -- there is a link on the top of this page -- and upload your zip file.

Keep a local copy for your records.