My Linked List

A ***linked list*** is a truly abstract data type (ADT). While an array is a list of elements ordered by index (in a contiguous section of the computer's memory), a linked list is just a series of ***nodes*** that contain references to the next node in the list. Linked lists are self-referential data structures and can be confusing at first!

Unlike previous labs, you will be writing a (simplified) Linked List-type class *first*. To understand how a linked list works, you must first understand how it works (an amusingly recursive conundrum).

My Linked List

1. In this lab, you will be creating a "singly" linked list, meaning that each node has a reference to the next node only. A "doubly" linked list is a list where each node has a reference to the previous AND next nodes. While there are a few conveniences that could be included to make working with a linked list easier, this lab is intentionally simplified to help you understand the concept.

    a. There is a PowerPoint on Canvas that may help – in addition, there are tons of resources on linked lists on the web.

2. Create a class **MyLinkedList.java**. Begin the class with the following:

    Instance variables

    - `ListNode head` – a reference to the first element in the list (more info on what ListNode is below)

    Inner class

```
private class ListNode
{
	int val; //does a private inner class need private instance variables?
	ListNode next;

	public ListNode(int val) { this.val = val; }

	@Override
	public String toString() { return "" + this.val; } //for printing / debug
}
```

    Woah – a class inside a class. Everything you learned in CS1 was a lie.

    A private inner class works like a private method – it's available only to the declaring class, and does something useful for that class (something that users of your class' objects don't necessarily need access to). The only difference is that private inner classes can define their *own* variables and methods, and produce objects.

    ListNode objects will serve as a bundle of some data (in this case, an integer* called `val`), and a reference to the next ListNode object in the list. This is essentially a helper

class, only for the use of the MyLinkedList class. Because it's located inside another class, the ListNode type will only be available to MyLinkedList objects.

> **If you're aware of* **generics***, it is suggested you NOT try to make your linked list generic as you learn a new type of data structure. We will discuss generics soon enough!*

Methods

- `MyLinkedList()` – default constructor, sets `head` to `null`
- `MyLinkedList(int val)` – parameterized constructor, begins the list with an initial node (with a null `next` reference)

3. Complete the `void add(int newVal)` method.

   a. Just like an ArrayList, this method will append an element to the end of the list. Unlike an ArrayList, though, there is no convenient indexing! This may be a good time to start panicking.

   b. An "empty list" will be simply a `head` node with a `null` reference. If `head` is `null`, set it equal to a new ListNode object, given `newVal`.

   If `head` *isn't* `null`, you must traverse the list (generally using a while loop and checking for a `null` *next* reference), beginning at the `head`, until you reach the end (a `null` reference). When you reach the end, simply add a reference to a new ListNode to the last node's `next`.

```
/* A temporary ListNode object that references the "current" node in the list,
as you iterate through, will be helpful.  You don't want to be changing the
value of the head ListNode! */
```

4. Complete the `boolean contains(int target)` method.

   a. This method returns true if `target` can be found in the list. In other words, if it can be found as the data member (`val`) in one of the nodes in the list.

   b. With ordinary indexed arrays, this process is trivial. With linked lists, you must again traverse the list, beginning at the `head`, and iterate through all the nodes until you reach the end (a `null` reference).

5. Complete the `int get(int index)` method.

   a. This method should return the element at the specified index. Throw a new IndexOutOfBoundsException if the `index` argument is out of bounds.

   b. With ordinary arrays, the `get` method happens in O(1) time. Linked lists will require O(n) time for retrieval, as you must traverse all elements to find a specific index.

6. Complete the `int indexOf(int target)` method (should return -1 if `target` can't be found).

7. Complete the `void set(int newVal, int index)` method. You shouldn't need to make a new ListNode object.

    a. Don't allow clients to set values at indexes outside the bounds of the list.

    b. This method will set (replace) the value in the list at the specified index with `newVal`. Iterate through the collection until you reach the desired index.

8. Complete the `int size()` method, that will return the number of elements in the list. Note that traditionally this is done with a `size` instance variable that is updated dynamically, similar to what you did in your MyStack class. This method will require another traversal of the list.

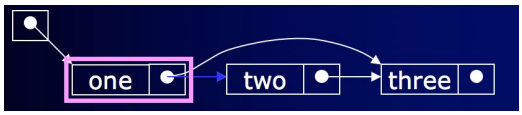
    a. Complete the method `int sizeRecursive()` that implements the `size` method recursively. You'll need a parameter for this, the "current" node.

9. Complete the `boolean isEmpty()` method, which should return true if the list has a size of zero. This shouldn't require any iteration!

10. Complete the `int remove(int index)` method.

    a. This method should remove (and return) the element at the specified index. Remember – in a singly linked list, each node has only a reference to the *next* node in the list. Think about the node you will need to "stop" at to remove an element at a specific index.

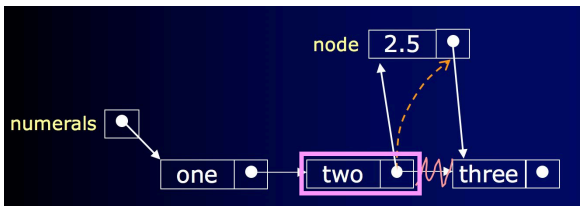
    b. The references should be changed to act as if the node that should be removed doesn't exist. This is in contrast with regular arrays, which require all elements to be moved when an element is removed – and is one of the big advantages of a linked list. The graphic below illustrates this process quite nicely:



    Even though the object with value "two" is still referencing the next object, it will no longer be part of the list. When the Java compiler notices that this object is no longer being used, the object will be destroyed (this process is known as "garbage collection").

11. Complete the `void add(int newVal, int index)` method.

    a. This method will insert `newVal` at the specified `index` into the list, "shifting" the remainder of the list's elements down. You will need to update node references accordingly. The image below illustrates this process quite well:



12. Complete an overridden `toString` method in this form: `[1, 2, 3, 4]` (use `[]` for an empty list). You must again traverse the list, until you reach a `null` reference (the end of the list).

13. Use the **Runner.java** file provided in the lab folder to test your code (download it and copy/paste or drag/drop into your project). Your output should match that in the **"output.txt"** file, in the lab folder.

14. A common idiom in linked lists is to, in addition to the `head` node, maintain a reference to the *last* element in the list, such that appending can happen without having to traverse the entire list to reach the end. In other words, as elements are appended to the end of the list, the reference to the last node simply changes to a reference to a new node (that will have a null `next` reference, indicating the end of the list). The `head` and `tail` references will be the same if the list has one element.

    Add a `ListNode tail` instance variable that will store a reference to the last element in the list, and refactor your code to take advantage of this added convenience.

    a. You will no longer have to traverse the list to add an element, simply add a new ListNode reference to `tail.next`, and change the value of `tail` to reference the new "last" node.

    b. Test that your code still works with the addition of the `tail` functionality.

15. In addition to the `tail` functionality, further improve your linked list class by maintaining the size of the list with an `int size` instance variable (similar to what you did when creating array-backed list and stack classes), rather than traversing the list and counting the elements (if you didn't already).

    As elements are added to the list, increment `size`. Decrement `size` with an element is removed. The `size` method is now O(1), instead of O(n) - much better!

(Optional) Iterable linked list

The convenience of the for-each loop is made possible by the Iterable<T> interface. Types that can be iterated should implement this interface (the for-each loop makes an Iterator behind the scenes, which it uses to iterate over a collection).

Add the code to make your MyLinkedList class Iterable. Iterable has just one method that returns an Iterator (something that implements the Iterator<T> interface).

When done, check that your MyLinkedList class can now be properly iterated with a for-each loop.