UNIVERSITY OF VIRGINIA, DEPARTMENT OF COMPUTER SCIENCE

Lab 04 - Linked Lists

Dr. Mark R. Floryan

August 9, 2018

1 PRE-LAB

This week, you will be writing a generic doubly-linked list data structure in Java. Your summary:

1. Download the starter code and import the project into Eclipse
2. Implement the LinkedList.java class
3. Verify your implementation using the provided tester class
4. **FILES TO DOWNLOAD:** linkedLists.zip
5. **FILES TO SUBMIT:** LinkedList.java

1.1 LISTITERATOR.JAVA

Once you've imported the provided code into Eclipse, you may want to start on ListIterator.java (though you don't have to). A ListIterator is an object that points to one element in your linked list, and provides methods to grab the element at that index, move the iterator forward one position, backward one position, etc. The provided tester uses this iterator class

to test your code, and so it much be implemented correctly. The methods you will be asked to implement are:

```
    /**
     * These two methods tell us if the iterator has run off
     * the list on either side
     */
    public boolean isPastEnd();
    public boolean isPastBeginning();

    /**
     * Get the data at the current iterator position
     */
    public T value();

    /**
     * These two methods move the cursor of the iterator
     * forward / backward one position
     */
    public void moveForward();
    public void moveBackward();
```

1.2 LinkedList.java

Next, implement all of the methods in LinkedList.java. This class will implement the provided List interface, which is duplicated for your convenience here. Your task is to implement each of these methods.

```
public interface List<T> {

    /**
     * Returns the size of this list, i.e., the number
     * of nodes currently between the head and tail
     * @return
     */
    public int size();

    /**
     * Clears out the entire list
     */
    public void clear() ;

    /**
```

```
16          * Inserts new data at the end of the
            * list (i.e., just before the dummy tail node)
18          * @param data
            */
20      public void insertAtTail(T data);

22      /**
            * Inserts data at the front of the
24          * list (i.e., just after the dummy head node
            * @param data
26          */
        public void insertAtHead(T data);
28
        /**
30          * Inserts node such that index becomes the
            * position of the newly inserted data
32          * @param data
            * @param index
34          */
        public void insertAt(int index, T data);
36
        public T removeAtTail();
38
        public T removeAtHead();
40
        /**
42          * Returns index of first occurrence of
            * the data in the list, or -1 if not present
44          * @param data
            * @return
46          */
        public int find(T data);
48
        /**
50          * Returns the data at the given index, null if
            * anything goes wrong (index out of bounds, empty list, etc.)
52          * @param index
            * @return
54          */
        public T get(int index);
56  }
```

In addition, there are a few Linked List specific methods that you need to implement. They

are all of the methods that involve a ListIterator in some way. The are enumerated below:

```
/**
 * Inserts data after the node pointed to by iterator
 */
public void insert(ListIterator<T> it, T data);

/**
 * Remove based on Iterator position
 * Sets the iterator to the node AFTER the one removed
 */
public T remove(ListIterator<T> it);

/* Return iterators at front and end of list */
public ListIterator<T> front();
public ListIterator<T> back();
```

1.3 Testing Your Implementation

Once you implement this methods, you can test them by running the main method in the provider ListTester.java class. This simple tester will execute the methods in your list and compare them to those in Java's built in LinkedList to make sure the results are as expected.

One strategy might be to implement your linked list methods in the order that they are tested. That way, you can see the tester say "this method is correct" before moving on to the next one.

Notice that we expect your Linked List to be a generic class, meaning any type of Object can be stored within your Linked List.

2 IN-LAB

The goal of this in-lab is to continue practicing with Linked Lists in a laid back environment. As usual, you will:

1. Get into small groups of two

2. The TAs will present you with some programming challenges regarding Linked Lists. These will not be graded, but you are required to attend and to participate.

3. You may think about the challenges before lab below if you'd like, but we highly recommend that you not solve them ahead of time.

4. The TAs will give you time to solve each problem and lead you in sharing solutions with one another.

2.1 LINKED LIST CODING CHALLENGES

The TAs will lead you in going through the following challenges. It is ok if you do not get through each of these.

1. Write a method in the LinkedList class called reverse(). This method should completely reverse the Linked List so that the first element is now the last, the second is second to last, etc. Make sure to adjust the head and tail references as well.

2. Write the find(T itemToFind) method for a Linked List, but write it **recursively**.

3. Write a small code segment that takes Linked List and splits into two Linked Lists, with half the elements in one, and the other half in the other. If the number of elements is odd, then the extra element should go in the first list.

4. Write a method, called removeDuplicates(), for a Linked List the assumes the list is sorted in increasing order, and removes all duplicates from the list. You may only traverse through the list once.

5. Write a method called merge(ListNode a, ListNode b) that takes pointers to the head of two Linked Lists (a and b) that are each sorted in increasing order. The method should combine a and b to create one long list that is in sorted order. You may not create any additional List Nodes in your method.

6. Suppose you try to print out your Linked List and the code is taking a very long time. You wonder if the list has a loop in it or if the list is just very long. Write a method that determines whether or not a Linked List is actually a loop or not. The method should return a boolean.

3 POST-LAB

The goal of this post-lab is to write a report comparing the runtimes of the various methods of your LinkedList class. The requirements for this post-lab are VERY simular to last week.

1. Write some test code to time the various methods of your Linked List
2. Run a small experiment timing each method
3. Write a report summarizing and analyzing your findings
4. **FILES TO DOWNLOAD:** None
5. **FILES TO SUBMIT:** PostLabFour.pdf

3.1 TIMING YOUR CODE

For this postlab, we'd like you to write a tester that executes each of your methods in your Linked List and times how long each takes. We are doing this to gain some intuition about which operations are slow and which operations are fast. You should consider the following when timing your code:

- Each method, if invoked only once, will run VERY quickly. Thus, instead we are going to invoke each method of our Linked List many times (you can adjust this number) to get a better measurement. We want the total time for each operation to be at least 1 second.
- Likewise, if methods are invoked on very small lists (e.g., finding an element in a list of size 5), then the operations will be VERY fast. Thus, make sure to make the lists large enough until you start to see a slowdown. Part of your investigation involves figuring out how big these lists must be before a slowdown is observed.
- Make sure you are ONLY invoking the method of interest when testing that method. Don't inadvertantly invoke find() while testing insertAtTail() as that will throw off your measurement.

To actually time the code itself, Java provides a system call that returns the number of milliseconds between now and January 1, 1970. The line of code is:

```
/* In java.lang.System */
public static long currentTimeMillis();
```

To use this in order to time some code, one can make this method call to get a timestamp, then run some code, then take another timestamp (using currentTimeMillis). The difference between the two timestamps is the number of milliseconds that have passed between the two method calls. So, you can test a method by doing something like:

```
import java.lang.System;

//Ready to begin measuring
int t1 = System.currentTimeMillis();

/* Lot's of code that runs a method on my list a bunch of times here */

//Difference in now and first time stamp is total time taken
int time = System.currentTimeMillis() - t1;
```

For this experiment, you should time your Linked List calling each method i times on a list of size n. You may play around with i and n until you find numbers that work well, but they should be the same for each method. You should test the following methods:

```
/* You will test these methods: */

public void insertAtTail(T data);
public void insertAtHead(T data);
public void insertAt(int index, T data);
public void insert(ListIterator<T> it, T data);
public T removeAtTail();
public T removeAtHead();
public T remove(ListIterator<T> it);
public int find(T data);
public T get(int index);
```

3.2 Report

Summarize your experiment and your findings in a report. Make sure to adhere to these general guidelines:

- Your submission MUST BE a pdf document. You will receive a zero if it is not.
- Your document MUST be presented as if submitted to a professional publication outlet. You can use the template posted in the course repository or follow Springer's guidelines for conference proceedings.

- You should write your report as if it is original novel research.
- The grammar / spelling / professionalism of this document should be sound.
- When possible, do not use the first person. Instead of "I ran the code 60 times", use "The code was executed 60 times...".

In addition to the general guidelines above, please follow the following rough outline for your paper:

- **Abstract**: Summarize the entire document in a single paragraph
- **Introduction**: Present the problem, and provide details regarding the two strategies you implemented.
- **Methods**: Describe your methodology for collecting data. How many hands, how many executions, how you averaged things, etc.
- **Results**: Describe your results from your execution runs.
- **Conclusion**: Interpret your results. Which methods were fast and which were slow? Did this surprise you? Does this align with the theoretical runtimes of those methods? How large did the lists need to get before you witnessed a slowdown?

Lastly, your paper MUST contain the following things:

- A table (methods section) summarizing the different experimental groups and how many execution runs were done in each group.
- A table (results section) summarizing each experimental group and the averages / std. dev. for each (as well as any other data you decided to collect).
- Some kind of graph visualizing the results of the table from the previous bullet.