Lab 2: The Circular LL and the Josephus Problem

Due Saturday, September 18, 2010 by 11:59PM EDT

Learning Objectives

Upon completion of this lab

• You will have a decent knowledge of how to implement the List interface to create your own Linked List Class (although java comes with a LinkedList class our goal is to keep the concepts separated from language)
• You will understand how to simplify and solve a seemingly complex problem, like Josephus problem, using a circular linked list
• You will also understand how to write testers to work with LL’s of different types

Concepts Covered

The following concepts are covered in this lab

• Interfaces – List and Comparable
• Command line arguments
• Circular Singly Linked Lists
• Debugging with Testers

Description of the Josephus Problem

This problem goes back to Josephus Flavius, a 1st century Roman historian. The story is slightly bloody.

41 rebels are trapped in a cave, surrounded by enemy troops. They decide to commit suicide: they line up in a circle and systematically kill every other one, going around and around, until only one rebel is left -- who supposedly kills himself (fat chance).
Who is the lone survivor?

Clearly, we can represent the rebels by a list of numbers \{1,2,\ldots,41\} and we can simulate the demise of the corresponding rebel by manipulating this list. For example, we could just mark the dead rebels by switching item \textit{i} to -\textit{i} (a standard hack, and a true abuse of negative numbers). Actually, it will be more interesting to generalize slightly: let us deal with the problem for an arbitrary number \textit{n} of rebels. So, our original list is \{1,2,\ldots,n\}, and we have to make a \textit{n}-1 deletions in the right place to find the survivor.

A good implementation will sometimes differ from a verbatim translation of the original problem. For example, marking the deceased rebels in the list without actually removing them turns out to be a bad idea: we will have to skip over the dead bodies. It is easier to actually remove items from the list. However, it is still quite complicated to do the necessary bookkeeping (we have to skip over the next element, and we have to be careful when we reach the end of the list). If we use an array to simulate this problem, the things can be quite complicated. We need to remove every other element in the array to remove dead bodies. Besides there is a huge cost to filling the holes in the array since we have to shift elements to fill the holes. Here is a simple idea that dissolves all these difficulties, once and for all. Let’s think about a linked list to do this problem. Not just a linked list, but a circular linked list. One step in our simulation will be to

\textit{Rotate the list to the left by one position, and then delete the first element.}

This does implement the every-other-guy protocol of the rebels. Make sure you understand why. Also note that the description of the problem is really a bit ambiguous: we might also delete the first element, and then rotate. As it turns out, the results are slightly easier to describe in the other version. Here is a run of the simulation for \textit{n} = 16 rebels. The sequence of survivors is shown after each death, but rotated a bit. In the end, rebel number 1 is the lucky one.

```
1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
3  4  5  6  7  8  9 10 11 12 13 14 15 16 1  3
5  6  7  8  9 10 11 12 13 14 15 16 1  3  5
7  8  9 10 11 12 13 14 15 16 1  3  5  7
9 10 11 12 13 14 15 16 1  3  5  7  9
11 12 13 14 15 16 1  3  5  7  9
13 14 15 16 1  3  5  7  9
15 16 1  3  5  7  9
11 13 15 1
1  3  5  7
11 13 15
1  5  9
9
1
1
```
And here are the results of the simulation for all rebel numbers up to 100. The x-axis is the number of rebels, and the y-axis is the lone survivor.

Clearly, there is a lot of regularity here. Increasing n by 1 increases the number of the survivor by 2 for a while, but then a reset occurs and the survivor number goes back to 1. It is tempting to find a nice, concise description for this effect, but we will not pursue the analysis at this time. Sometime in your life you will learn that there is a ridiculously simple way to calculate the survivor by manipulating the binary expansion of the number of rebels. In any case, we need an excuse to implement a linked list, a circular linked list and then use that to solve the josephus problem. This will demonstrate why choosing the right data structure is important to simplify the problem and solve it.

\Starter Files\n
The project is organized into the following six files:

- List.java // List interface is implemented by CircularLL class
- node.java // the basic node class
- CircularLL.java // This is a class that implements the List interface
- CLLIntTester.java // this is the tester for a circular LL of Integers
- CLLStringTester.java // this is the tester for a circular LL of Strings
- josephus.java // Driver program for Josephus
The node.java defines Node type and methods of basic building block Node, that can be used to build a circular linked list as the main data structure. Such a list consists of nodes containing the data element (an Integer object), plus a pointer next, pointing to the next node, respectively. Moreover, the last node in the list points to the first. Thus, the nodes really form a linked circle. The list header points to the first node.

CircularLL.java is a class that contains methods necessary to solve the Josephus problem and you need to complete them as well as complete the driver program to produce the desired output.

One advice of thought is to test your circular LL class before using it in the josephus program. In fact solving Josephus problem is minor part of the implementation. This way of testing LL class independent of the problem assures you that your LL methods are working properly. Once you test this, solving the Josephus problem is easy.

The input/output conventions are as follows. Program can be managed from a simple command line menu.

**Method to implement**

The class CircularLL implements List interface methods. You must implement the following methods in the CircularLL class.

```java
public void clear(); // clear the list by removing each node
public int size(); // returns the number of nodes in the list
public void add(Comparable c); // add a node to the end of the circular list. End node is connected to head
public String toString();
public boolean contains(Comparable c); // returns true if list contains c
public boolean isEmpty(); // returns true if list is empty
public Object remove(int index); // remove the node at index. List starts from index 0
public boolean remove(Comparable c); // remove the object c. Return false if c is not in list
public void reverse(); // reverses the CLL

public void rotate(int n); // rotate the list by n elements. If n is positive do clockwise rotation, and if n is negative do counter clockwise rotation.
```

You may add any other private methods if you wish. Note that data type inside a node object is a Comparable object. For example, you can create an Integer object from an int n using

```java
Integer N = new Integer(n); See Integer class in the API to see which methods are useful in completing this assignment.
```

You also need to complete the driver program, josephus.java. It must take command line arguments, process them, and call the correct circular LL methods to solve the problem. You may not necessarily use all of the methods implemented from the List interface to solve the josephus problem, but implementation of all List methods is part of the assignment.
Testing CircularLL class methods in Eclipse

Note that there are 3 files that contain main program. They all use the CircularLL class. You can only run one main program at a time. So if you would like to test CLLIntTester then right click on the file in eclipse and choose run as application. You can repeat the same process in other files with main.

Testing Josephus problem in Eclipse

To test Josephus problem, you would need some command line arguments as defined below. The following cases must be tested. (see command line section below for descriptions)

1. java josephus 41
2. java josephus 10 20
3. java josephus –a 10
4. java josephus –a 10 –s 5

These are the only cases to be tested. In case 1, there is one command line argument, args[0], cases 2 and 3 contains two command line arguments, args[0] and args[1] and case 4 contains 3 command line arguments args[0], args[1] and args[2]. Your Josephus main program must first determine how many command line arguments are given and then parse through things to process them. Your program must also handle error cases where not enough and wrong number of arguments are given.

Using Command Line arguments in Eclipse

Using command line arguments args[0], args[1] etc when running from command line is easy (see below). But in eclipse you have to do few things. Go to

Run ➔ Run Configurations

And enter command line arguments required for each main file in the Program Arguments box (each command line argument must be in a separate line)
Command Line Section

Testing CircularLL in command Line

This section is given for you to learn how to run programs from command line arguments. You don’t need to do both, but it is good to know that the programs you develop in eclipse can also be run from command line (without an IDE). This will allow us to run java programs in any platform you like. Here are some of things you can test.

Testing CircularLL class with ints and Strings in command line

- javac CircularLL.java CLLIntTester.java
- java CLLIntTester
- javac CircularLL.java CLLStringTester.java
- java CLLStringTester

Testing Josephus Problem in command line

You can complete the program using eclipse. The following instructions are for command line testing only.

1. Solve Josephus problem for some n (assume that Josephus is the class file). You need to complete driver program Josephus.java to handle the following cases.

   > java josephus 41

   41 19
2. Solve Josephus problem for a range of values.

> java josephus 10 20

<table>
<thead>
<tr>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>12</td>
<td>9</td>
</tr>
<tr>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
</tr>
<tr>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>19</td>
<td>7</td>
</tr>
<tr>
<td>20</td>
<td>9</td>
</tr>
</tbody>
</table>

3. Solve the Josephus problem for a single value, but display the state of the line after each shooting. Here the option “-a” indicates that we need to show all steps in the process.

See Josephus.java for more details.

> java josephus -a 10

1 2 3 4 5 6 7 8 9 10
3 4 5 6 7 8 9 10 1
5 6 7 8 9 10 1 3
7 8 9 10 1 3 5
9 10 1 3 5 7
1 3 5 7 9
5 7 9 1
9 1 5
5 9
5

4. Start solving the Josephus problem, but stop when you reach a certain list size. Here –s 5 indicates that when the list reaches size 5, we need to output the remaining list. The 10 is the size of the original list.

See Josephus.java for more details.

> java josephus -a 10 -s 5

1 3 5 7 9

Note: No other command line argument combinations will be tested. You just need to write the Josephus file to cover the above cases only.

---

**Downloading Files**

Download your files from course web
Handing in your Solution

Your solutions should be in the form of .zip files. When we grade your solution we will unzip the folder and test the code. You can submit your entire eclipse folder named yourID_Lab2.zip to Bb ➔ tools ➔ dropbox

Note: We will not grade any other file name and you will receive a zero for the assignment. No exceptions to this rule.

Grading Criteria

The following grading criterion is strictly enforced.

1. A program that does not compile - 0 points
2. A program that is 24 hours later – max of 50% of the grade
3. A program that is more than 48 hours late – 0 points

100 points total, distributed as follows:

- Circular linked list class works with Integer Tester : 50 points
- Circular linked list class works with String Tester : 10 points
- Solving the Josephus : 30 points
- Style: Commenting and indenting: 10

Make sure that your program compiles and runs properly. As usual, take the 10 style points seriously; poorly commented spaghetti code is not acceptable.