Section 9:
More Linked Lists

Pair up with anyone who is agreeable to pairing up with you, pick the first driver, and let’s get to work!

PART I: Another Linked-List Tracing Exercise

Being able to trace through the execution of a linked-list program is an essential skill for a programmer, which is why we’re giving you another tracing exercise. This time, you need to trace through the execution of a method to debug it.

1. On the class web page you’ll find a program creatively named PartI09.java, and another named LLNode.java. Load both into DrJava. (PartI09.java requires LLNode.java.)

2. PartI09.java contains a main() method that tests deleteOdds(), which is intended to delete the odd-positioned nodes from a linked list. For example, if the list is:

```
+---+ +-----+---+ +-----+---+ +-----+---+ +-----+---+
|   | --+-->| A | --+-->| B | --+-->| C | --+-->| D | / |
```

deleteOdds() will remove the nodes containing A and C, as they are the first and third nodes, leaving a two-node list with B’s and D’s nodes.

The code for deleteOdds() appears to be complete, but has a bug, as you can see by compiling and running PartI09. Do that, and study the output. List n has n nodes, each holding the values 1 through n, with List 0 being the empty list.

3. The testing output for deleteOdds() should be:

```
Testing deleteOdds:
List 0 Before:  
List 0 After:  
List 1 Before: 1  
List 1 After: 2  
List 2 Before: 1 2  
List 2 After: 2 3  
List 3 Before: 1 2 3  
List 3 After: 2 3 4  
List 4 Before: 1 2 3 4  
List 4 After: 2 3 4 5  
List 5 After: 2 4  
```

Trace through the execution of deleteOdds() on one or more of the sample lists, until you find the cause of the bug. Be sure to draw pictures of the list(s) and of the values of the variables so that you can keep track of what’s happening. Remember what the bug is; your SL will want to know!

4. Fix deleteOdds(), and recompile and rerun PartI09. Repeat until the testing output is correct.

✔ CHECKPOINT 1 Raise your hand. Your SL will come over and ask about the deleteOdds() bug, and will also check that your deleteEvens() works.

(Continued ... )
PART II: Building an Ordered Linked-List of Exam Objects

To build a singly-linked-list of Exam objects in order, we need (a) a node class and (b) a minimal linked-list class. We’ll give you the node class, but you have to complete the linked-list class. Specifically, LLNode objects are used by ExamList to construct a singly-linked-list of Exam objects.

1. You should already have LLNode.java (from Part I). For this part you’ll also need Exam.java, ExamList.java, and PartII09.java from the class web page. Load them all into DrJava.

2. Review LLNode.java. Yes, you used it in Part I, but you need to know the names of the methods.

3. Next, study Exam.java. An Exam node stores the last name and section letter of a student. Note that Exam implements Comparable, and has a toString() method that displays the state in the form X:n, where X is the section letter and n is the name. For example, a student named Williams in Section G would be returned by toString() as G:Williams.

4. Now look at ExamList.java. It’s partially complete; the instance variables, the constructor, a getter, append(), and toString() are supplied.

   Question: We have a getter for occupancy, but not for the other instance variable. Why is it a bad idea to have a getter for it?

5. Time to complete ExamList. First: Write the body of the prepend() method. It is to add the Exam object to the front of the list, and return the updated occupancy of the list. We’ve studied and used prepend() methods in other programs; we hope this step will be easy!

6. Before worrying about insert(), test your implementation of prepend(). Compile and run the PartII09 program. The output from the prepend() tests are in the middle; ignore the output from the insert() tests at the bottom. When your prepend() is producing the correct output, you can move on.

7. Now for the big one: Write insert(). If you’ve completed Program #8, you know what needs to be done: The method needs to be able to handle four situations: (a) inserting into an empty list, (b) inserting ahead of the first node of the list, (c) inserting between two existing nodes, and (d) inserting after the last node. It’s possible to combine (a) and (b), and also (c) and (d), to create a method with less code, but all it needs to do is work in those four situations. Write it!

   Helpful Hint #1: Don’t try to write the whole method before you test it! Instead, write the code to handle inserting into an empty list, and test that (PartII09 tries that situation first). Then add code to insert ahead of the first node, and test that. (PartII09 tries that situation next). Testing frequently helps isolate the errors to one small portion of code, making debugging much less irritating.

   Helpful Hint #2: Got a bug? Draw pictures of list situations and trace the code, as you did for Part I.

8. When your insert() handles all four cases and successfully passes all of the tests, you’re ready for the checkpoint.

   ✔ CHECKPOINT 2 Raise your hand. Your SL will come over and check that prepend() and insert() are complete.

(Continued …)
PART III: Convert an SLL to a DLL

In last Wednesday’s section activity, you were asked to write a method that turned a singly-linked list (SLL) into a circularly-linked list. In this part, you’ll convert an SLL into a doubly-linked list (DLL), whose nodes reference the same data objects as do the nodes of the SLL.

1. On the class web page is the file DLLNode.java, which you will use to create the nodes for your new DLL. Load it into Dr. Java.

2. Compare it to the LLNode class you used in the first two checkpoints. What are the differences?

3. Also on the web page is the file ExamDLLList.java. It’s not complete. Guess who’s going to complete it? (Good guess! :-)) Here is what you need to do:

(a) Complete the gnirtsOt() method. ExamDLLList.java has a toString() method; gnirtsOt() does the same thing but puts the data in the string in reverse order.

(b) The second constructor builds the DLL from the given SLL. We’ve started the code of this constructor, but you have to finish it. For each SLL node, you’ll have to create a DLL node that references the same data object, and link it into the growing DLL.

That’s it; get to it!

4. We have two questions about the constructor you just wrote:

   • Question #1: The constructor accepts a reference to the first LLNode of the SLL. Why doesn’t the constructor accept a reference to an ExamList object?

   • Question #2: How else might we provide the constructor with the data necessary to build the DLL?

5. Get the PartIII09.java program from the class web page, load it into Dr. Java, and compile and run it. If you wrote both gnirtsOt() and the constructor correctly the first time, you should be very proud of yourselves. Otherwise, you’re like the rest of us, and have some debugging to do: Fix either or both of those methods until you pass PartIII09.java’s testing.

   ✔ CHECKPOINT 3 Raise your hand. Your SL will come over and check both your answers and your test output.

PART IV: Clean Up!

1. Log out of your computer.

2. Pick up your papers, writing implements, cell phones, trash, etc.

3. Push in your chair(s).

   ✔ CHECKPOINT 4 Raise your hand. Your SL will come over and use his or her X-ray vision to mutate any remaining germs into worthy arch-enemies.

You’re free to go! But, if you have time, we recommend that you use it to work on the programming assignment.