

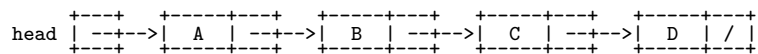
Section 10: A Three-Course Meal of Linked Lists
--

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 keep giving you tracing exercises. 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 `PartI10.java`, and another named `LLNode.java`. Load both into DrJava. (`PartI10.java` requires `LLNode.java`.)
2. `PartI10.java` contains a long `main()` method that tests two methods, `deleteOdds()` and `deleteEvens()`. These methods are to delete the odd-positioned and even-positioned nodes from a linked list. For example, if the list is:



`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. `deleteEvens()` would do the opposite.

The code for `deleteOdds()` appears to be complete, but has a bug, as you can see by compiling and running `PartI10`. 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 as shown below:

```
Testing deleteOdds:
List 0 Before:
List 0 After:
List 1 Before: 1
List 1 After:
List 2 Before: 1 2
List 2 After: 2
List 3 Before: 1 2 3
List 3 After: 2
List 4 Before: 1 2 3 4
List 4 After: 2 4
List 5 Before: 1 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 `PartI10`. Repeat until the testing output is correct.
5. `deleteEvens()` is supposed to work as described above, but is obviously only a stub method at this time. Copy and paste the body of `deleteOdds()` into this stub, and adjust the code so that it deletes the even-positioned nodes.

(Continued ...)

- Verify that `deleteEvens()` is working by compiling and running `PartII10`. You should see this output if it is working correctly:

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

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

PART II: Creating a Comparable Class

Many Java API classes implement the *Comparable* interface, which means that objects from such a class can be compared, using the `compareTo()` method, to determine how their states compare (smaller, larger, or equal).


In this part, you and your partner will complete a class named `Exam`. Objects of this class hold just a student's last name and the letter of their section. You will use `Exam` in Part III to create an ordered linked list of `Exam` objects in a sorted order, much as the CSc 127B staff creates of exams after we grade them.

- Open a new file in DrJava named `Exam.java`.
- We've completed most of `Exam.java` for you. All you have to do is write:
 - A `toString()` method that returns the state of the object 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`.
 - The `int compareTo(Exam)` method required by `Comparable`. Here's how it needs to work: If the section letters of the two `Exam` objects are different, the objects are ordered by the ASCII values of the section letters. Thus, any `Exam` object from section A is less than and `Exam` object from section B. When the section letters are the same, the ordering is by the name. For example, if there are two students in section H named Baker and Andrews, the Andrews `Exam` object is less than Baker's. If the sections and names match in both objects, they are equal.

If you don't remember how `compareTo()`'s returned values are chosen, review the `Comparable` API entry.

That's all you need to know; complete the `Exam` class!

- On the class web page, in the Section 10 area, is a program named `PartIII10.java`. Its job is to test your `Exam` class. Load it into DrJava, compile it, and run it. When you're getting the output shown in the comment at the top of `PartIII10.java`, you're ready to move on.

 **CHECKPOINT 2** Raise your hand. Your SL will come over and verify that your `Exam` class is complete.

(Continued ...)

PART III: 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. On the class web page are the files `LLNode.java` (in the Part I area), `ExamList.java`, and `PartIII10.java`. Load them all into DrJava (`LLNode.java` is likely still loaded).
2. Review `LLNode.java`. Yes, you used it in Part I, but you need to know the names of the methods.
3. 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?

4. 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. If you watched and understood my `prepend/insert` video from Friday, this should be easy. (It's post #265 on Piazza.)
5. Before worrying about `insert()`, test your implementation of `prepend()`. Compile and run the `PartIII10` program. You can ignore the output from the `insert()` tests at the bottom. 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.
6. Now for the big one: Write `insert()`. If you watched Friday's video, 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 (`PartIII10` tries that situation first). Then add code to insert ahead of the first node, and test that. (`PartIII10` tries that situation next). Testing frequently helps isolate the errors to one small portion of code, making debugging much less irritating.


Helpful Hint #2: Draw pictures of list situations, like you did for Part I!

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

 **CHECKPOINT 3** Raise your hand. Your SL will come over and check that `insert()` is complete.

PART IV: Clean Up!

1. Log out of your computer; pick up your papers, writing implements, cell phones, trash, etc.; push in your chair(s).

 **CHECKPOINT 4** Raise your hand. Your SL will come over and use his/her trained nose to sniff out evidence of decaying germs.