CSC 115: Fundamentals of Programming II
Assignment #2: Linked Lists

Due date

Sunday, October 7, 2018 at 23:55 pm via submission to conneX.

How to hand in your work

Submit the requested files through the Assignment #2 link on the CSC 115 conneX site. Please make sure you follow all the required steps for submission – *including confirming your submission!*

Learning outcomes

When you have completed this assignment, you will have learned:

- How to create an implementation of *TaskList* using a *reference-based linked list*.
- How to *use reference values* including the *null* value.
- How to debug Java code that makes extensive use of ref-based variables.

Task Scheduling

Problem description

Modern computer operating systems (such as Mac OS X, Linux, and Windows 10) are designed and constructed around several main concepts. One concept is the *task* (sometimes called a *process* or *thread*). When you use the Java compiler, it runs as a task; when you start up a web browser, it runs as a task. There are also many tasks running "in the background". In fact, some tasks generate additional tasks in order to complete some larger, more involved computation. Every task has many properties, including those known as *task priority* and *task number*.

Since an operating system (or *OS*) is organized around the running of tasks, there exists a central algorithm that schedules tasks, i.e., it chooses the next task to run when the current task must halt or pause. Making this choice is relatively easy: the scheduler simply takes the next available task having the highest priority, and the task is then given a chance to run on the computer. A *task list* is a data collection used to arrange waiting tasks in an order suitable for the OS scheduler. That is:

- When a task is created, it is placed into the task list at a position appropriate for the task's priority and arrival time.

- When the OS must choose a task to execute, it takes the one at the head of the task list.

In this assignment we will not be modifying an existing operating system (!) but we will play with an implementation of operations needed for a *Task List ADT*. The data structure to be used for the implementation is a *referenced-based linked list*.

Examples

Imagine the following four tasks arrivals (also called four *TA* events) that occur in the order shown:

```
(10, 212)
(12, 100)
(10, 198)
(3, 104)
```

The first number in each line corresponds to that task's *priority*, and the second number corresponds to the task *number*. Here our first task has a priority of 10 and a task number of 212. We want our list organized such that higher priority tasks are closer to the head, but tasks with the same priority are in order of arrival time (i.e., earlier tasks are closer to the head). If the four TA events are the first ones in our computer, then the resulting task list will be:

```
{ (12, 100), (10, 212), (10, 198), (3, 104) }
```

When the scheduler chooses a task for execution (an event denoted as *SC*) it removes the first task from the list – in fact, what the scheduler needs most is the task number of the task at the head of the list. Therefore the next task to be scheduled is:

```
100
```

and the task list after the SC event is:

```
{ (10. 212), (10, 198), (3, 104) }
```

Sometimes an OS decides a task must *deleted* before the task has a chance to execute. This means removing the task from the list (a *TD* event). Suppose task number 198 is to be deleted. Once deletion occurs, our resulting task list is:

```
{ (10, 212), (3, 104) }
```

Knowing the number of tasks in the task list (or asking for its status, or an *ST* event) can be important for other parts of the operating system. Since the number of tasks is simply the list's length, we can see there are currently two tasks in the system. If we add three more TA events:

```
(10, 85)
(4, 314)
(2, 101)
```

then the resulting list is:

```
{ (10, 212), (10, 85), (4, 314), (3, 104), (2, 101) }
```

and an ST event would result in the following value:

```
5
```

Task Scheduling

For this assignment you will complete *TaskListLL.java* which as this will be your ref-based linked-list implementation of the *TaskList* interface. Your *TaskListLL* will be used by code that simulates the TA, TD, SC and ST actions described above. (The simulator code is contained in *A2test.java* and used specifically in the last four tests.) That is, the simulator will use the following completed routines in your implementation of *TaskListLL*:

- *insert*: used by TA
- *remove*: used by TD
- *removeHead*: used by SC
- *getLength*: used by ST
- *isEmpty*: used by the simulator to control the event loop

The required behavior of these methods is described in *TaskList.java*.

Several other files are also provided to you:

- *TaskList.java*: A representation of the data for a *Task* along with *equals()* and *compareTo()* methods.
- *TaskListNode.java*: To be used by code within *TaskListLL*.
- *TaskListArray.java*: A completed implementation of the *TaskList* interface but using an array to store tasks.
- *A2test.java*: The tester program for this assignment. When run without any arguments, it used *TaskListLL* for testing; if at least one argument (for example, "abc") is provided, then *TaskListArray* is used instead.

Here is a run of *A2test* that uses *TaskListArray*:

```
$ javac A2test.java
$ java A2test abc
Testing of 'TaskList' class (basic)
Passed test: 1
Passed test: 2
...
Running task simulation 3
Passed test: 39
Running task simulation 4
Passed test: 40
```

And here a run of *A2test* that uses *TaskListLL* as it is distributed with this assignment.

```
$ java A2test
Testing of 'TaskList' class (basic)
ECHO getLength()
Failed test: 1 at line 216
```

(Note that you should remove all of the "ECHO" *println* statements in your completed implementation of *TaskListLL*.)

As with assignment #1, when we evaluate your work it is the first failed test that will matter most for us (i.e., we do not cherry-pick tests by ignoring earlier failed tests while accepting later passing tests). Tests after the first failed test are themselves considered to have failed.

There are 40 tests and your code is expected to pass them all.

File to submit:

- *TaskListLL.java*

Grading scheme

Requirement	Marks
TaskListLL.java compiles without errors or warnings	2
Code passes the first set of test cases (1 to 16)	4
Code passes the second set of test cases (17 to 22)	3
Code passes the third set of test cases (23 to 28)	4
Code passes the fourth set of test case (29 to 36)	5
Code passes the four simulations (37 to 40)	2
Total	**20**

Note: Code submitted must be written using techniques in keeping with the goals of the assignment. Therefore passing a test is not automatically a guarantee of getting marks for a test (i.e., your solution must not be written such that it hardcodes results for specific tests in *A2test.java* yet would be unable to work with similar tests when different data is used).

In order to obtain a passing grade for the assignment, you must satisfy at least the first four requirements by passing all of their test cases.