COMPUTER SCIENCE 12B (FALL TERM, 2017)
PROGRAMMING IN JAVA

EXTRA CREDIT PROGRAMMING ASSIGNMENT

Due Date: Thursday December 14th, 11:55pm

This assignment focuses on implementing a linked list. Turn in the following file using the link on the course website:

• `AssassinManager.java` – A class that manages a game of Assassin.

You will need the support files `AssassinMain.java`, `AssassinNode.java`, and `names.txt`; place these in the same folder as your program or project. The code you submit must work properly with the unmodified versions of the provided files.

Program Details

"Assassin" is a game often played on college campuses. Each person playing has a particular target that he/she is trying to "assassinate." Generally "assassinating" a person means finding them on campus in public and acting on them in some way (e.g. saying "You're dead," squirting them with a water gun, or touching them). One of the things that makes the game more interesting to play in real life is that initially each person knows only who they are assassinating; they don't know who is trying to assassinate them, nor do they know whom the other people are trying to assassinate.

Assassin Rules

1. You start out with a group of people who want to play the game
2. A circular chain of assassination targets (called the "kill ring" in this program) is established.
3. When someone is assassinated, the links need to be changed to "skip" that person.

Example Game of Assassin

Let's walk through an example with five people playing: Carol, Chris, Jim, Joe, Sally. We might decide Joe should stalk Sally, Sally should stalk Jim, Jim should stalk Carol, Carol should stalk Chris, and Chris should stalk Joe. In the actual linked list that implements this kill ring, Chris's next reference would be null. But, conceptually we can think of it as though the next person after Chris is Joe, the front person in the list.

Here is a picture of this "kill ring":

front -> Joe -> Sally-> Jim -> Carol -> Chris ->

Then, suppose Sally assassinates Jim. Sally needs a new target, so we give her Jim's target: Carol. The kill ring becomes:

front -> Joe -> Sally-> Carol -> Chris ->

If the first person in the kill ring is assassinated, the front of the list must adjust. If Chris kills Joe, the list becomes:

front -> Sally-> Carol -> Chris ->

In this assignment, you will write a class `AssassinManager` that keeps track of who is stalking whom and the history of who killed whom. You will maintain two linked lists:

- a list of people currently alive (the "kill ring") and
- a list of those who are dead (the "graveyard").

As people are killed, you will move them from the kill ring to the graveyard by rearranging links between nodes. The game ends when only one node remains in the kill ring, representing the winner.

A client program called `AssassinMain` has been written for you. It reads a file of names, shuffles the names, and constructs an object of your class `AssassinManager`. This main program then asks the user for the names of each victim to kill until there is just one player left alive (at which point the game is over and the last remaining player wins). `AssassinMain` calls methods of the `AssassinManager` class to carry out the tasks involved in administering the game.

Assassin Manager

You must use our `AssassinNode` classs provided on the course website without modification.

In the lectures we have been looking at nodes of type `ListNode` that have just two fields: a field called data of type `int` and a field called `next` that points to the next value in the list. The `AssassinNode` class has three fields. The first two are fields for storing data called `name` and `killer` (they are used to store the name of a player and the name of the person who killed that player). The third field is called `next` and it serves the same purpose as the next field in the `ListNode` class.

`AssassinManager` should have the following fields:

1. a reference to the front node of the kill ring
2. a reference to the front node of the graveyard (null if empty)

Note that a requirement of this assignment is that you have **exactly** these two fields and **no others**.

`AssassinManager` should have the following constructor:

`public AssassinManager(List<String> names)`
This constructor should initialize a new assassin manager over the given list of people. Note that you should not save the list parameter itself as a field, nor modify the list. Instead, you should build your own kill ring of list nodes that contains these names in the same order. If the list is null or empty, you should throw an `IllegalArgumentException`. For example, if the given list contains ["John", "Sally", "Fred"], your initial kill ring should represent that John is stalking Sally who is stalking Fred who is stalking John (in that order). You may assume that the names are non-empty, non-null strings and that there are no duplicates.

`AssassinManager` should also implement the following methods:

`public void` **`printKillRing`**`()`
This method should print the names of the people in the kill ring, one per line, indented by **four** spaces, as "X is stalking Y ". If the game is over, then instead print "X is stalking X". For example, using the names in the example game above, the output is:

```
>>     Joe is stalking Sally

>>     Sally is stalking Jim

>>     Jim is stalking Carol

>>     Carol is stalking Chris

>>     Chris is stalking Joe
```

If the game is over and Chris is the winner, so Chris is the only name in the kill ring, the output is:

```
 >>     Chris is stalking Chris
```

`public void` **`printGraveyard`**`()`

This method should print the names of the people in the graveyard, one per line, with each line indented by four spaces, with output of the form "name was killed by name". It should print the names in the opposite of the order in which they were killed (most recently killed first, then next more recently killed, and so on). It should produce no output if the graveyard is empty.

For example, using the names from above, if Jim is killed, then Chris, then Carol, the output is:

```
>>     Carol was killed by Sally

>>     Chris was killed by Carol

>>     Jim was killed by Sally
```

`public boolean` **`killRingContains`**`(String name)`

This method should return true if the given name is in the current kill ring and false otherwise. It should ignore case in comparing names; so, "salLY" should match a node with a name of "Sally".

`public boolean` **`graveyardContains`**`(String name)`

This method should return true if the given name is in the current graveyard and false otherwise. It should ignore case in comparing names; so, "CaRoL" should match a node with a name of "Carol".

`public boolean` **`isGameOver`**`()`

This method should return true if the game is over (the kill ring has one person) and false otherwise.

`public String winner()`
This method should return the name of the winner of the game, or null if the game is not over.

`public void kill(String name)`
This method should record the assassination of the person with the given name, transferring the person from the kill ring to the front of the graveyard. This operation should not change the relative order of the kill ring (i.e. the links of who is killing whom should stay the same other than the person who is being killed). This method should ignore case in comparing names. A node remembers who killed the person in its killer field, and you must set the value of this field. You should throw an `IllegalStateException` if the game is over, or throw an `IllegalArgumentException` if the given name is not part of the kill ring. If both of these conditions are true, the `IllegalStateException` takes precedence.

You will be graded in part by how efficient your code is, especially your `kill` method. Your methods should run in *O(n)* time where *n* is the number of nodes in the list. In particular, **you may NOT use nested loops** in any method in order to get full credit on this assignment.

Constraints

This is meant to be an exercise in linked list manipulation. As a result, you must adhere to the following rules while implementing `AssassinManager`:

- You may not construct any arrays, `ArrayLists`, `LinkedLists`, `Stacks`, `Queues`, or other data structures; you must use list nodes. You may not modify the list of `Strings` passed to your constructor.

- If there are *n* names in the list of `Strings` passed to your constructor, you should create exactly *n* new `AssassinNode` objects in your constructor. As people are killed, you have to move their node from the kill ring to the graveyard by changing references, without creating any new node objects.

- Your constructor will create the initial kill ring of nodes, and then your class may not create any more nodes for the rest of the program. You are allowed to declare as many local variables of type `AssassinNode` (like current from lecture) as you like. `AssassinNode` variables are not node objects and therefore don't count against the limit of *n* nodes.

You are not required to turn in any testing code for this assignment, but you should write some of your own testing code. `AssassinMain` requires every method to be written in order to compile, and it never generates any of the exceptions you have to handle, so it is not exhaustive.

Circular Lists

Some students try to store the kill ring using a "circular" linked list (where the list's final element stores a next reference back to the front). It is significantly more difficult to write bug-free code using a circular list. There is no need to use a circular list for this assignment, because you can always get back to the front via the fields of your `AssassinManager`. If you feel strongly that you want to use a circular list, you may, but it will make the program significantly more difficult to write so we discourage it.

Sample Log of Execution

Your program should reproduce the format and behavior demonstrated in this log. Note that you may not exactly recreate this scenario because of the shuffling of the names that AssassinMain performs.

```
>> Welcome to the CS12b Assassin Manager

>>
>> What name file do you want to use this time? names3.txt
>> Do you want the names shuffled? (y/n)? n
>>
>> Current kill ring:
     >>      Athos is stalking Porthos
     >>      Porthos is stalking Aramis
     >>      Aramis is stalking Athos
>> Current graveyard:
>>
>> next victim? Aramis
>>
>> Current kill ring:
     >>      Athos is stalking Porthos
     >>      Porthos is stalking Athos
>> Current graveyard:
     >>     Aramis was killed by Porthos
>>
>> next victim? Athos
>>
>> Game was won by Porthos
>> Final graveyard is as follows:
     >>      Athos was killed by Porthos
     >>      Aramis was killed by Porthos
```

Grading

You will be graded on:

- **External Correctness:** The output of your program should match exactly what is expected. Programs that do not compile will not receive points for external correctness.
- **Internal Correctness:** Your source code should follow the stylistic guidelines shown in class. Also, remember to include the comment header at the beginning of your program. Specifically: part of your grade will come from appropriately utilizing recursive backtracking to implement your algorithm as described previously. We will also grade on the elegance of your recursive algorithm; don't create special cases in your recursive code if they are not necessary or repeat cases already handled. Redundancy is another major grading focus; some methods are similar in behavior or based off of each other's behavior. You should avoid repeated logic as much as possible. Your class may have other methods besides those specified, but any other methods you add should be private.

- **One-on-one interactive grading:** By the end of the day that the assignment is due, please make an appointment with your TA for an interactive 10-15 minute grading session. You will receive an email notifying you of the name of the TA who has been assigned to you for this assignment with further instructions on setting up the appointment. (You will be meeting with a different TA for each assignment). One-on-one interactive grading will help you improve your programming skills and avoid repeating mistakes from one assignment to the next.