Assignment 3

Create a class that maintains the top scores for a video game using a linked list. For this project, you will create a ***linked list*** of `GameEntry` ***objects***.
The provided classes are: the GameEntry class (You can read more about the GameEntry class here: http://cs.slu.edu/~goldwasser/dsaj/docs/index.html?dsaj/arrays/GameEntry.html), and the Node class. Read them carefully. Notice that the `Node` class contains a `GameEntry` field!

Part A. Singly Linked List. Start by creating a singly linked list to use as your data structure for this task. In addition to the `head` field, you will also maintain a `tail` field. Include the below standard linked list features/methods in your class, which you should call `HiScoresLL`. *Code up all the method headers exactly as given below.*

Basic features of `HiScoresLL`:

- `head` and `size` fields, which we learned about in class and reading
  - A `tail` field that references the last node in the list (akin to the `head` field) Keep in mind:
    - This field must be used/maintained/updated during operations like removing and adding nodes in case the last (or only) node is removed or a new node is being added to the end of the list.
    - Keep in mind the end of the list could be the same as the beginning of the list if there is only one item in it.
    - It is initially set to null, when the list is empty, just like the `head` field.

- A constructor method – creates a new empty linked list

```
/**
 * Constructor that creates an empty list
 */
public HiScoresLL() { }
```

- A `display()` method – outputs all the high score entries in the list. Remember, `GameEntry` objects come with a `toString()` method (see its definition in the `GameEntry` class) so you can directly print a `GameEntry` object `e` simply with `System.out.println(e)`.

```
/**
 * Prints out all the game entries in the linked list
 */
public void display(){ }
```

- An `addFirst(Node v)` method – adds the node `v` to the front of the list

```
/**
   * Add a node to the head of the list
    * @param v
    * the Node object to be added

    */
    public void addFirst(Node v){ }
```

- A `removeFirst()` method – removes a node from the front of the list

```
/**
      * Removes the first node and returns it,
       * this method assumes the list is non-empty
       * @return
       * the Node that was removed

       */
       public Node removeFirst(){ }
```

- An `addLast(Node v)` method – adds the node `v` to the end of the list. Note: now that we have a reference to the tail of the list, this method can be implemented in constant time!

```
/**
      * Add a node to the tail of the list
       * @param v
       * the Node object to be added

       */
       public void addLast(Node v){ }
```

Code up all the method headers with preceding comments exactly as given above. These headers are formatted in a specific way to allow for some nifty automatic creation of .html (web page) documentation for your class. This web page documentation for your program can be instantly generated by simply using the command: `javadoc HiScoresLL.java` at the command prompt. After running this command, you'll see the .html document appear in the same directory as your .java files.

For each method, be sure to test your work by adding test code to a `main` method in a `Test` class. From reading the code for `GameEntry` and `Node`, you can see that one way to create and add a high score to a test list would be:

```
HiScoresLL hsList = new HiScoresLL();
      GameEntry ge = new GameEntry("Grand Master",1000); Node v = new
      Node(ge, null);
      hsList.addFirst(v);
      hsList.display();
```

Alternatively, you can combine the middle three lines into one by writing:

```
hsList.addFirst(new Node(new GameEntry("GM",1000), null));
```

After you've implemented and tested these basic features, you're ready to add the two methods that allow someone to use your class to maintain a list of high scores for a video game. In implementing these next two methods, you may make calls to the basic list manipulation methods you created above.

- An `add(GameEntry e)` method, which only needs to work properly when the items in the list are already ordered by high score as expected

```
/**
    * Assuming the list of game entries is in decreasing order by score,
    * this method creates a Node with the given GameEntry e, and then
        * inserts it in the appropriate spot in the list.
        * @param e
        * the GameEntry object to be added to the list

        */
        public void add(GameEntry e){ }
```

- `remove(int i)` – removes the i^{th} node, returning the corresponding game entry object

```
/**
    * Removes the node at position i in the list
    * (emulating an array index)
        * @return
        * the GameEntry of the removed node
        * or null if position i is invalid

        */
        public GameEntry remove(int i){ }
```

Part B (Bonus 20 points). Doubly Linked List. Create a second program that achieves the same goal as your first one, but uses a doubly-linked list rather than a singly-linked list. You will need to first create a `DNode` class, which you can base on the `Node` class, that will have an extra field (called `prev`) for the backward pointer. `DNode` is the class you will use to build your doubly linked list. The name of your doubly-linked list class should be `HiScoresDLL` and it should be saved in a file called `HiScoresDLL.java`.

For this program, you needn't write all the "basic functionality" methods that you did for the singly-linked list. (This is due to the more general nature of the header and trailer fields in a doubly-linked list, which we will learn about in class and reading. No special cases to check this time!) So, after creating the fields and the constructor method, you can go directly to writing the `add(GameEntry e)` and `remove(int i)` methods.

Warning: you might be tempted to try copying and pasting from the methods you wrote above, then modifying the code. However, these methods in a doubly-linked list class are *significantly* different, and more streamlined. You should write them from scratch rather than confuse things by trying to reuse old code. *If you find yourself creating code that seems more unwieldy or complicated than the corresponding code for Part A, then you are going in the wrong direction. Part B's code, done well, should be much* ***more*** *compact and elegant, not* ***less****. (No special cases needed!)*

One caveat: now that we have a doubly-linked list, allowing us to walk forward from the front *or* backward from the end with equal ease, the `remove(int i)` method should be written such that it makes the fewest number of pointer hops needed to get to the game entry at index `i`.

Part C (2%). Final Challenge: devise a simple (or not-so-simple) game that the player can play from the basic command-line interface that tracks players' high scores. The rules of the game are completely up to you, so I encourage you to (at least at first) keep it extremely basic, then you can make it more interesting if you have extra time. The only requirement of the game is that after each time the game is played, you must use the high scores class that you wrote in Part A or Part B to maintain a list of high scores. The game should start with a main menu that prompts the user to either "a) play a new game," "b) see high scores," or "c) shut down game." Option c) simply ends the program. After each time a player plays your game, update the high scores appropriately by simply calling the methods you created above. The game should then loop back to the main menu. If anyone ever chooses "c) shut down game," which ends the whole program, the high scores will be lost, which is fine for now. But as long as the game is up and running, the high scores list should be maintained properly. I highly recommend that you get your game idea approved first so that I can give you a sense of any feasibility issues that may arise. The simplest way to take user input from the console is probably to use java's built-in Scanner class. Here is an example from your textbook of how to use it. Compile and run it to see how it works. Here are some other links on how to use scanner objects if you need more guidance:

- http://www.cs.williams.edu/~jeannie/cs136/scanner.pdf (note on this document: apparently the `nextString()` method doesn't work, so you have to use `next()`)
- http://webclass.superquest.net/apjava/JavaNotes%20Reference/notes-java-2006-04-12/notes-java/examples-introductory/console/console-input-scanner.html

For all these i/o examples, you should compile and run them to see how they work. Make sure it's clear to you what each line of code is doing!

Submission. Make an appointment to discuss your project with me.

Extra Credit (1%) For some bonus credit, learn to do file I/O so that you can write the high scores out to a text file each time the program is exited, then read them in again the next time the program runs, thereby maintaining the high scores list from one run of your game to the next.