Programming Assignment #4: Garland

COP 3330, Fall 2019

Due: Sunday, November 17, *before* 11:59 PM

Abstract

In this programming assignment, you will code up a data structure that uses linked lists to represent strings. You will implement methods that add, retrieve, and remove strings from that data structure. In doing so, you will gain experience working with linked lists, and you will strengthen your working knowledge of Java in general and references in particular.

Important note: In your assignments, you can use any code I've posted in Webcourses, as long as you leave a comment saying where that code came from. Of course, you cannot use code posted by other professors, and you should never incorporate or refer to code from online resources or from other individuals.

Deliverables

Garland.java

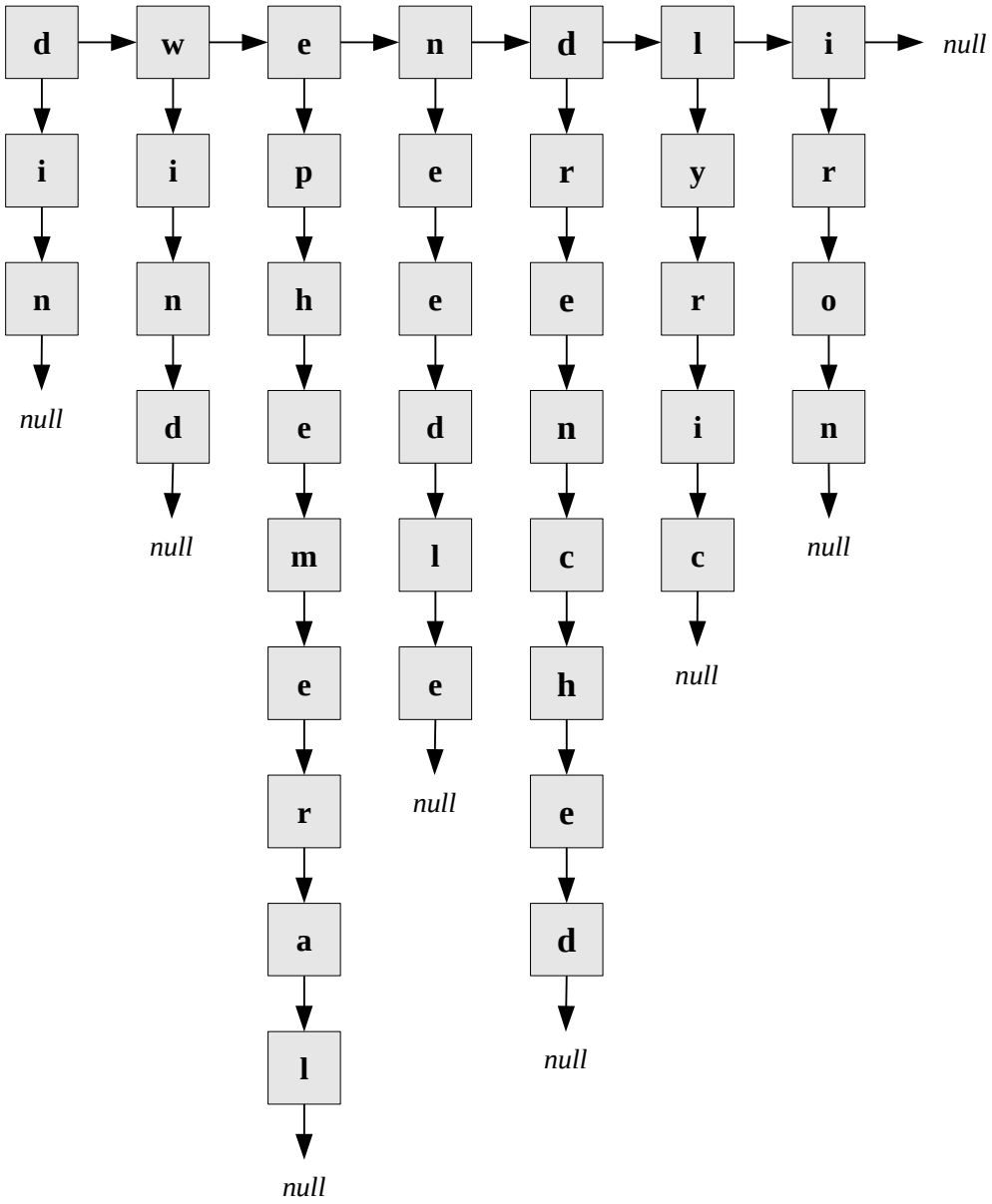
Note! The capitalization and spelling of your filename matter!

Note! Code must be tested on Eustis, but submitted via Webcourses.

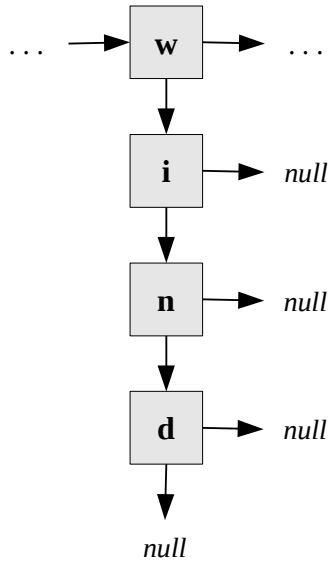
1. Overview

In this assignment, you will create a garland, which is a sort of two-dimensional linked list for storing strings. Each node in this data structure will contain a single character, as well as two references: a *next* reference that is used to reach the next string in the garland and a *down* reference that is used to reach the next character within a particular string.

For example, the following garland contains seven strings (“din”, “wind”, “ephemeral”, “needle”, “drenched”, “lyric”, and “iron”). The top-left node in the diagram (with the character ‘d’) is the head node of this entire garland data structure. That node has a *down* reference that takes us to the next character in “din”, as well as a *next* reference that takes us to the ‘w’ node in our representation of “wind”. Subsequently, that ‘w’ node has a *down* reference that takes us to the next character in “wind”, as well as a *next* reference that takes us to the first character in “ephemeral”.



Note that the *next* field is only used across the top layer of nodes in the garland. For example, in the nodes representing “wind” in the garland above, only the ‘w’ node has a *next* reference that actually leads to another node. The ‘i’, ‘n’, and ‘d’ nodes all have their *next* references set to *null*. So, the more complete diagram for that particular section of the garland looks like so:



Note that every node in the garland contains a single character. The garland has no means by which we can represent an empty string, since an empty string would have zero nodes. Any string in a garland will have at least one node, which means it must have at least one character.

Note also that the layer of *next* references across the top of the garland (which connects the first letter of each string in the garland) will always be terminated with a *null* reference, and the chain of *down* references used to connect the characters in a given string will also always be terminated with a *null* reference.

The *Node* class you must use for this assignment (included in a file named *Node.java*) is as follows.

```
public class Node
{
   char data;  // Each node holds a single character.
   Node next;  // Reference to next string's first node (provided we are at
               // the first node in some string; if not, this is null).
   Node down;  // Reference to node with the next character in this string.

   // This is the only available constructor for this class.
   Node(char data)
   {
      this.data = data;
   }
}
```

(*Warning!*) You **cannot** modify the *Node* class, you must **not** submit *Node.java* with your assignment, and you **cannot** add the *Node* class to your *Garland.java* file as a non-public support class.

2. Field Requirements

In your *Garland* class, you ***must*** have the following two fields, and they must both be ***private*** and ***non-static***:

```
public class Garland
{
   private Node head;  // Reference to the top-left node in this garland.
   private int size;   // The number of strings currently in this garland.
}
```

You may also have a *private Node tail* field (a reference to the garland's top-right node) if you'd like to use that to implement an optimally efficient *addString()* method (which performs tail insertion), but that is not required.

(*Warning!*) Your *Garland* class cannot have any additional fields beyond the ones listed in this section.

3. Method and Class Requirements

Implement the following methods in a public class named *Garland* in a file named *Garland.java*. Please note that these are all **public**. Some are static, and some are not. You must pay close attention to those modifiers in order to ensure that your code is compatible with our test cases. Note that these methods should not print anything to the screen unless explicitly called for by their method descriptions below.

You may implement helper methods as you see fit. Please include your name, the course number, the current semester, and your NID in a header comment at the very top of your source file.

(*Important!*) For all the methods below, be sure to update the garland's *size* field as appropriate, even if that isn't mentioned explicitly in the method descriptions.

`public static Node stringToLinkedList(String s)`

> **Description:** This method takes a single string (*s*) as its only input parameter and creates a linked list representation of that string (with one character per node, using the *down* references to link to each subsequent node in the list). All the *next* references in the nodes created by this method should be *null*.
>
> **Note:** Calling this method from *addString()* might help you simplify the logic in that method somewhat.
>
> **Note:** Please note that this is not an instance method. This method is static.
>
> **Return Value:** If *s* is the empty string or *null*, simply return *null*. Otherwise, return the head of the linked list created by this method.

`public static String linkedListToString(Node head)`

> **Description:** This method takes the head of a linked list and returns the string represented by that linked list. This method uses the *down* references in each node to move from character to character as it constructs the new string.
>
> **Note:** Please note that this is not an instance method. This method is static.

(*Important!*) Special Restriction: You cannot use string concatenation (the + operator) in this method! Recall that string concatenation is slow in Java. You must find a way to do this without concatenation.

Hint #1: (Highlight and/or copy and paste to reveal.)

Hint #2: (Highlight and/or copy and paste to reveal.)

Runtime Restriction: If you avoid string concatenation, the runtime for this method will be O(k) (where *k* is the length of the string).

Return Value: Return a string as described above.

`Garland()`

Description: Default constructor. No action required (since the fields are auto-initialized to the appropriate values: *null* for the *head* (and *tail*, if you decide to implement one) and 0 for *size*.

Return Value: This is a constructor method and therefore should not return a value.

`Garland(String [] strings)`

Description: Overloaded constructor. This method takes an array of strings and adds them, in the order given, to the garland. So, the array {“din”, “wind”, “ephemeral”, “needle”, “drenched”, “lyric”, “iron”} should give rise to the garland in the diagram on pg. 2 of this PDF.

Runtime Note: It’s possible to implement this method in O(nk) time (where *n* is the number of strings and *k* is the maximum string length). An implementation that repeatedly starts at the beginning of the garland and loops to the tail node to insert the next new string will have an $O(n^2k)$ runtime. The preference is for this method to have an O(nk) runtime, but that’s not strictly required.

Return Value: This is a constructor method and therefore should not return a value.

`public Node head()`

Description: Return the head of the garland (the head of the linked list). Note that it is common for getter method to simply share the name of the field they return – e.g., *head()* rather than *getHead()*.

Return Value: A reference to the head of the garland (the top-left node in the linked list).

`public int size()`

Description: Return the size of the garland (i.e., the number of strings it contains).

Runtime Restriction: This must have an O(1) runtime.

Note: For all the methods in this assignment, be sure to update the garland's *size* field as appropriate, even if that isn't mentioned explicitly in the method descriptions. That will allow you to simply return *this.size* from this method in O(1) time.

Return Value: An *int* indicating the number of strings currently contained in this garland.

`public void addString(String s)`

Description: This method takes a single string (*s*) as its only input parameter, creates a linked list representation of that string (with one character per node, using the *down* references to link to each subsequent node), and adds that new linked list to the tail end of the garland. If the garland is empty, this becomes the only string in the linked list, and the garland's *head* is set to the first node representing this string.

This method should be a no-op if *s* is either the empty string or *null*. (That means this method should effectively do nothing in those cases.)

Runtime Note: It's possible to implement this method in O(k) time (where *k* is the length of the string being inserted) if you maintain a tail reference in the *Garland* class (which is optional). If this method always has to start at the head of the garland and loop forward to find the tail before inserting each new string, the runtime for a single call to this method will instead be O(n + k) (where *n* is the number of strings already contained in the garland and *k* is the length of the string being inserted).

Return Value: Nothing. This is a *void* method.

`public Node getNode(int index)`

Description: This method takes a single integer (*index*) and returns the head node of the linked list at that index in the garland (i.e., the first node in the linked list representing some string). For this method, assume the strings in the garland are numbered 0 through (*size* - 1).

For example, in someone called *getNode(0)* on the garland depicted on pg. 2 of this PDF, the method would return a reference to the node containing the 'd' in "din". If someone called *getString(1)* on that same garland, the method would return a reference to the node containing the 'w' in "wind".

If this method receives an invalid index for the given garland (i.e., if *index* is negative, or if *index* exceeds (*size* - 1)), this method should simply return *null*.

Return Value: Return a *Node* reference as described above.

`public String getString(int index)`

Description: This method has the same behavior as the *getNode(int index)* method described above, except it returns the entire string represented by the linked list at the given index (rather than just the head node of that linked list).

(*Important!*) Special Restriction: You cannot use string concatenation (the + operator) in this method!

Recall that string concatenation is slow in Java. You must find a way to do this without string concatenation. See the hints in the *linkedListToString()* method for ideas on how to accomplish this.

Return Value: Return a string as described above.

```
public boolean removeString(int index)
```

Description: This method takes a single integer (*index*) and removes the string (i.e., the linked list) at that index from the garland. For this method, assume the strings in the garland are numbered 0 through (*size* - 1). If this method receives an invalid index for the given garland (i.e., if *index* is negative, or if *index* exceeds (*size* - 1)), this method should simply return *false*.

Return Value: Return *true* if this method successfully removes a string (i.e., a linked list) from the garland. Otherwise, return *false*.

```
public boolean printString(int index)
```

Description: Print the string stored at the given index in the garland, followed by a newline character. For this method, assume the strings in the garland are numbered 0 through (*size* - 1).

If this method receives an invalid index for the given garland (i.e., if *index* is negative, or if *index* exceeds (*size* - 1)), this method should print "(invalid index)" (without the quotes), followed by a newline character.

Return Value: Return *true* if the given index is valid. Otherwise, return *false*.

```
public void printStrings()
```

Description: Print all the strings in the garland. This method should print the strings in the same order in which they appear in the garland. Each string should be printed on its own line (followed by a newline character). If the garland is empty, simply print "(empty list)" (without the quotes), followed by a newline character. For examples of the expected output, see the test cases included with this assignment.

Note: It is **not** a good idea to repeatedly call *linkedListToString()* from this method. There is a more efficient way to do this.

Return Value: Nothing. This is a *void* method.

```
public static double difficultyRating()
```

Description: Return a double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult).

```
public static double hoursSpent()
```

Description: Return a realistic and reasonable estimate (greater than zero) of the number of hours you spent on this assignment.

4. Suggested Methods (Optional, but Helpful)

The following methods are suggested, but not required. I found they helped me keep my code looking clean and straightforward, and/or they helped simplify repetitive tasks that kept coming up across the required methods. I'll leave it to you to determine where it might be useful to have these methods.

`private static int getListLength(Node head)`

> **Description:** This method takes the head of a linked list representing a string and returns the number characters in that string (i.e., the number of nodes linked via *down* references).
>
> **Return Value:** Returns an *int* as described above.

`private boolean validIndex(int index)`

> **Description:** Takes a single integer (*index*) and returns *true* if that integer corresponds to a valid index in the garland, given the number of strings it currently contains. Otherwise, this method returns *false*.
>
> **Return Value:** Returns a *boolean* as described above.

`private void printString(Node head)`

> **Description:** This method takes the head of a linked list as its only input parameter and prints the string represented by that linked list. This method uses the *down* references to move from character to character. If *head* is null, this method simply prints "(null)" (without the quotes), followed by a newline character.
>
> **Return Value:** Nothing. This is a *void* method.

5. Compiling and Running All Test Cases (and the *test-all.sh* Script!)

The test cases included with this assignment are designed to show you some ways in which we might test your code and to shed light on the expected functionality of your code. We've also included a script, *test-all.sh*, that will compile and run all test cases for you.

Recall that your code must compile, run, and produce precisely the correct output on Eustis in order to receive full credit. Here's how to make that happen:

1. At the command line, whether you're working on your own system or on Eustis, you need to use the *cd* command to move to the directory where you have all the files for this assignment. For example:

   ```
   cd Desktop/garland_assignment
   ```

   Warning: When working at the command line, any spaces in file names or directory names either need to be escaped in the commands you type, or the entire name needs to be wrapped in double quotes. For example:

```
cd garland\ assignment
```

```
cd "garland assignment"
```

It's probably easiest to just avoid file and folder names with spaces.

2. To compile your program with one of my test cases:

```
javac Garland.java TestCase01.java
```

3. To run this test case and redirect the program's output to a text file:

```
java TestCase01 > myoutput.txt
```

4. To compare your program's output against the sample output file I've provided for this test case:

```
diff myoutput.txt sample_output/TestCase01-output.txt
```

If the contents of *myoutput.txt* and *TestCase01-output.txt* are exactly the same, *diff* won't print anything to the screen. It will just look like this:

```
seansz@eustis:~$ diff myoutput.txt sample_output/TestCase01-output.txt
seansz@eustis:~$ _
```

Otherwise, if the files differ, *diff* will spit out some information about the lines that aren't the same.

5. I've also included a script, *test-all.sh*, that will compile and run all test cases for you. You can run it on Eustis by placing it in a directory with *Garland.java* and all the test case files and typing:

```
bash test-all.sh
```

Super Important: Using the *test-all.sh* script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting. Note that this script might have limited functionality on Mac OS systems or Windows systems that aren't using the Linux-style bash shell.

6. Transferring Files to Eustis

When you're ready to test your project on Eustis, using MobaXTerm to transfer your files to Eustis isn't too hard, but if you want to transfer them using a Linux or Mac command line, here's how you do it:

1. At your command line on your own system, use *cd* to go to the folder that contains all your files for this project (*Garland.java*, *Node.java*, *test-all.sh*, the test case files, and the *sample_output* folder).

2. From that directory, type the following command (replacing `YOUR_NID` with your actual NID) to transfer that whole folder to Eustis:

```
scp -r $(pwd) YOUR_NID@eustis.eecs.ucf.edu:~
```

Warning: Note that the `$(pwd)` in the command above refers to your current directory when you're at the command line in Linux or Mac OS. The command above transfers the *entire contents* of your current directory to Eustis. That will include all subdirectories, so for the love of all that is good, please don't run that command from your desktop folder if you have a ton of files on your desktop!

7. Style Restrictions (*Same as Previous Assignments*)

Please conform as closely as possible to the style I use while coding in class. To encourage everyone to develop a commitment to writing consistent and readable code, the following restrictions will be strictly enforced:

- ★ Capitalize the first letter of all class names. Use lowercase for the first letter of all method names.
- ★ Any time you open a curly brace, that curly brace should start on a new line.
- ★ Any time you open a new code block, indent all the code within that code block one level deeper than you were already indenting.
- ★ Be consistent with the amount of indentation you're using, and be consistent in using either spaces or tabs for indentation throughout your source file. If you're using spaces for indentation, please use at least two spaces for each new level of indentation, because trying to read code that uses just a single space for each level of indentation is downright painful.
- ★ Please avoid block-style comments: /* *comment* */
- ★ Instead, please use inline-style comments: // *comment*
- ★ Always include a space after the "//" in your comments: "// *comment*" instead of "//*comment*"
- ★ The header comments introducing your source file (including the comment(s) with your name, course number, semester, NID, and so on), should always be placed *above* your import statements.
- ★ Use end-of-line comments sparingly. Comments longer than three words should always be placed *above* the lines of code to which they refer. Furthermore, such comments should be indented to properly align with the code to which they refer. For example, if line 16 of your code is indented with two tabs, and line 15 contains a comment referring to line 16, then line 15 should also be intended with two tabs.
- ★ Please do not write excessively long lines of code. Lines must be no longer than 100 characters wide.
- ★ Avoid excessive consecutive blank lines. In general, you should never have more than one or two consecutive blank lines.
- ★ Please leave a space on both sides of any binary operators you use in your code (i.e., operators that take two operands). For example, use *(a + b) - c* instead of *(a+b)-c*. (The only place you do *not* have to follow this restriction is within the square brackets used to access an array index, as in: *array[i+j]*.)
- ★ When defining or calling a method, do not leave a space before its opening parenthesis. For example: use *System.out.println("Hi!")* instead of *System.out.println ("Hi!")*.

- ★ Do leave a space before the opening parenthesis in an *if* statement or a loop. For example, use use *for (i = 0; i < n; i++)* instead of *for(i = 0; i < n; i++)*, and use *if (condition)* instead of *if(condition)* or *if(condition)*.
- ★ Use meaningful variable names that convey the purpose of your variables. (The exceptions here are when using variables like *i*, *j*, and *k* for looping variables or *m* and *n* for the sizes of some inputs.)
- ★ Do not use *var* to declare variables.

8. Special Restrictions (*Super Important!*)

You must abide by the following restrictions in this assignment. Failure to abide by certain of these restrictions could result in a catastrophic loss of points.

- ★ **(*New*)** You must use linked lists (and the provided *Node* class) in order to receive credit for this assignment. Attempts to produce correct output without actually using linked lists will result in zero credit.
- ★ **(*New*)** Your *Garland* class cannot have any member variables (i.e., fields) other than *head*, *tail* (optional), and *size*. Every other variable you create for this assignment must be declared within a method.
- ★ For this particular assignment, the only import statement you can have is for *java.util.Arrays*. We might automatically detect assignments with additional *import* statements and refuse to compile them for this particular assignment, resulting in zero credit.
- ★ You cannot use any features of Java or built-in classes that we have not covered in lecture. You can, however, use any methods from any class we've covered this semester, even if we haven't covered those specific methods explicitly. (For example, we've covered the *Arrays* class already in lecture. Even though we haven't talked about every single method within that class, you are welcome to use any of its methods as you see fit.) You cannot use Java's *StringBuilder* class, even if we cover it in lecture before the assignment deadline.
- ★ File I/O is forbidden. Please do not read or write to any files.
- ★ Do not write malicious code. (I would hope this would go without saying.)
- ★ No crazy shenanigans.

9. Deliverables (Submitted via Webcourses, Not Eustis)

Submit a single source file, named *Garland.java*, via Webcourses. The source file should contain definitions for all the required methods (listed above), as well as any helper methods you've written to make them work.

Be sure to include your name, the course number, the current semester, and your NID in a header comment at the very top of your source file.

10. Grading Criteria and Miscellaneous Requirements

> ***Important Note:*** When grading your programs, we will use different test cases from the ones we've released with this assignment, to ensure that no one can game the system and earn credit by simply hard-coding the expected output for the test cases we've released to you. You should create additional test cases of your own in order to thoroughly test your code. In creating your own test cases, you should always ask yourself, "What kinds of inputs could be passed to these methods that don't violate any of the input specifications, but which haven't already been covered in the test cases included with the assignment?"

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

80%	Passes test cases with 100% correct output formatting. This portion of the grade includes tests of the *difficultyRating()* and *hoursSpent()* methods.
20%	Adequate comments and whitespace and sound programming practices. To earn these points, you must adhere to the style restrictions set forth above. We will likely impose huge penalties for small deviations, because we really want you to develop good style habits in this class. For some methods, we might also check that you're using good functional decomposition and/or the DRY principle ("don't repeat yourself," also referred to in class as "never repeat the same code twice"). This portion of the grade might also include credit for having a header comment at the top of your source code with your name and NID.

(*Important!*) Your program must be submitted via Webcourses, and it must compile and run on Eustis in order to be eligible for credit. Programs that do not compile and run on Eustis will automatically receive zero credit.

11. Final Thoughts (*Abbreviated*)

- ★ Please be sure to submit a *.java* file (not a *.class* file, and certainly not a *.doc* or *.pdf* file).
- ★ After submitting to Webcourses, please consider downloading your source code, re-compiling it, and re-testing it in order to ensure that you uploaded the correct version.
- ★ Please consider removing your *main()* method from *Garland.java* if you have one and then double checking that your program still compiles before submitting.
- ★ Please do not articulate a *package* in your *Garland.java* file.
- ★ Please be aware that the test cases included with this assignment are not comprehensive. Please be sure to create your own test cases and thoroughly test your code.
- ★ Sharing test cases with other students is allowed, as long as those test cases don't include any solution code for the assignment, but you should challenge yourself to think of edge cases before reading other students' test cases.

Start early! Work hard! Ask questions! Good luck!