CSE 373, Winter 2013
Homework Assignment #3: The Amazing Deque (25 points)
Due Friday, February 1, 2013, 11:30 PM

This program focuses on implementation of a linear collection called a *deque* using an array as an internal data structure. Turn in three files named `MazeSolver.java`, `ArrayDeque.java`, and `mymaze.txt` from the Homework web page.

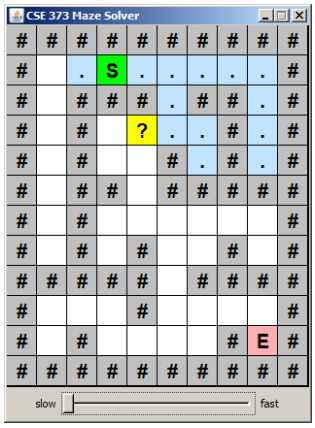
You will also need to download the support files `Main.java`, `Maze.java`, `GraphicalMaze.java`, `Deque.java`, and various input text files from the Homework section of the course web site; place them in your Java project and add to it. If you are using Eclipse, place .java files in your project's `src/` folder and .txt files in the project root folder (the parent of `src/`). We also provide a 'stub' version of the other files on the web site as a starting point if you like.

A set of provided data files to use as input is also posted on the Homework section of the web site. We do not guarantee that the main program is an exhaustive test. Perform additional testing of your own before you submit your program.

Part A (Maze Solver) Description:

input file maze1.txt:

```
##########
#  S     #
# ### ## #
# #    # #
# #  # # #
# ## #####
# #       #
# # #  # #
##### ####
#   #    #
# #    #E#
##########
```



A **deque** (pronounced "deck") is an ADT for a double-ended queue. It is a linear ordered collection of elements that allows the client to add and remove elements from the front and back of the queue in constant (O(1)) time. Unlike a normal queue, which allows insertion only at the back and removal only from the front, a deque allows both at either end.

This program consists of two parts. In the first part, you will use a deque to implement an algorithm to search for a path to escape from a 2-D maze. We are providing you with a `Maze` class representing the maze with various useful data and operations. You will write a class `MazeSolver` that implements the algorithm described by the following pseudo-code:

```
solve(maze):
    Keep a deque of squares to visit, initially empty.
    Put the maze's start location in the deque.
    Repeat the following until the deque is empty, or until we solve the maze:
        L := Remove the first location from the deque.
        If L is the maze's end location, then we have solved the maze.
        If we have already visited L before, or L is a wall, ignore it.
        Otherwise:
            Mark L as being visited.
            Add the neighbors of L (up 1, down 1, left 1, right 1) to your deque:
                If the neighbor is closer to the end location than L,
                add it to the front of the deque.  Otherwise add it to the back.
    If you end up with an empty deque and have not found the end, there is no solution.
```

The algorithm examines possible paths, looking for a path from the start to the end. It is more efficient if it examines paths that move closer to the end first. So it takes advantage of the nature of a deque by adding closer neighbors to the front of the deque, meaning that they will be processed sooner, and further neighbors to the end of the deque, meaning that they will be processed later. Maze locations are returned to you as Java `Point` objects that have a `distance` method you can use to see how far apart they are from each other. Note that we are using the standard screen coordinate system; "up" means negative *y* and "down" means positive *y*; "left" means negative *x* and "right" means positive *x*.

Part A (MazeSolver) Implementation Details:

Part A is the shorter and less challenging part of the assignment, intended to get you familiar with using deques and showing you a practical use of this collection to solve an interesting problem.

For this part you will use Java's provided `Deque` interface and `LinkedList` class from `java.util`. The methods of `Deque` that you are allowed to use are listed on the next page in the description of Part B. In Part B you will write your own implementation of a deque. Here is a brief example of using Java's provided `Deque` classes:

```
import java.util.*;   // using Java's provided Deque
...
Deque<String> example = new LinkedList<String>();
example.addLast("goodbye");
example.addFirst("hello");
```

The class you will implement in Part A is named `MazeSolver`. It has just one method; you must write it with exactly the header shown, so it can be called by the provided `Main` class. You **may define additional methods** in your class if you like, but they must be `private`. (You probably should not need any data fields to implement this behavior.)

`public boolean` **`solve`**`(Maze maze)`

In this method you will search for a path in the given maze from the start location to the end location using the deque-based algorithm described on the previous page, marking each square you visit along the way. If you find a path from the start to the end, you should return `true`; if not, you should return `false`.

This method is called by the provided `Main` class, passing you the maze for the input file the user indicates. If the maze passed is `null`, you should throw a `NullPointerException`. If the maze is non-null, you may assume that its state is completely valid, though some mazes do not have any successful path from the start to the end location.

Your `MazeSolver` interacts with our provided `Maze` class. That class has the following methods you can use:

`public Point` **`start`**`()`
`public Point` **`end`**`()`

Returns the starting and ending locations in the maze, respectively. A `Point` object has public `x` and `y` fields representing its position as well as some useful methods such as `distance`. See its Java API documentation.

`public int` **`width`**`()`
`public int` **`height`**`()`

Returns the *w*/*h* dimensions of the maze. The previous example maze1.txt is 10 characters wide and 12 tall.

`public boolean` **`isInBounds`**`(int x, int y)`

Returns `true` if the given *x*/*y* position is within the *w*/*h* bounds of the maze. Use this to avoid out-of-bounds indexes. (All of the other methods that accept *x*/*y* parameters throw an `IllegalArgumentException` if passed bad indexes.)

`public void` **`setVisited`**`(int x, int y)`

Marks the given *x*/*y* position as having been visited by your algorithm.

`public boolean` **`isVisited`**`(int x, int y)`

Returns `true` if the given *x*/*y* position has been previously marked as visited by a call to `setVisited`.

`public boolean` **`isWall`**`(int x, int y)`

Returns `true` if the given *x*/*y* position stores a wall of the maze (a `'#'` character).

`public String` **`toString`**`()`

Returns a text representation of the maze, which looks like its input file plus dots to mark any visited squares.

We also provide a `GraphicalMaze` class with the same methods as `Maze` plus an animated graphical user interface.

Part B (ArrayDeque) Description:

Part B is the more substantial part of the assignment where you practice implementing a non-trivial collection on your own. In this part you will write your own implementation of a deque. A deque can be implemented efficiently using either an array or a linked list of nodes; in our case you will use an array as the internal data structure.

A key aspect of a well-implemented deque is that its common operations such as adding and removing elements from both ends must be fast; they must run in O(1) average runtime. As you have learned, an unfilled array is efficient by nature for adding and removing at the end (highest indexes) but slow for removing from the front (index 0) because of the need to shift elements left by one to cover the hole left by the removed first element.

To get around this and implement the deque efficiently, your array will use a technique called a *circular buffer*. This means that as elements are added and removed, not only does the size (or "last meaningful index") of the array change, but if they are added/removed from the front, the start of the data (or "first meaningful index") also changes.

For example, suppose we have a 10-element circular array buffer representing a deque of strings. It is initially empty and then we add six elements to it as shown by calling `addLast` and passing A, then B, C, D, E, and then F:

index	0	1	2	3	4	5	6	7	8	9
value	A	B	C	D	E	F				

size 6 *front* 0

At this point the deque is [A, B, C, D, E, F]; a call to `peekFirst` would return A and `peekLast` would return F.

Now suppose the client calls `removeFirst` three times, which returns A, then B, then C. The deque's state is now the following. Notice that D-F did not shift to index 0 but remain in their original locations:

index	0	1	2	3	4	5	6	7	8	9
value				D	E	F				

size **3** *front* **3**

At this point the deque is [D, E, F]; a call to `peekFirst` would return D and `peekLast` would return F.

To use all available array space, a circular buffer wraps around to its other end if needed. Suppose the client calls `removeFirst` on D, then calls `addLast` six more times, adding G, H, I, J, K, and L. The deque's state is now:

index	0	1	2	3	4	5	6	7	8	9
value	**K**	**L**			E	F	**G**	**H**	**I**	**J**

size **8** *front* **4**

At this point the deque is [E, F, G, H, I, J, K, L]; a call to `peekFirst` would return E and `peekLast` would return L.

The wrapping occurs in both directions. Suppose we go back to our second diagram above that contained only D-F in indexes 3-5, and we called `addFirst` five more times, adding M, N, O, P, and Q. The deque's state would be:

index	0	1	2	3	4	5	6	7	8	9
value	**O**	**N**	**M**	D	E	F			**Q**	**P**

size **8** *front* **8**

At this point the deque is [Q, P, O, N, M, D, E, F]; a call to `peekFirst` would return Q and `peekLast` would return F.

If the deque becomes full, enlarge it by copying its data into a new array twice as large. You cannot simply copy over the old contents into the same indexes in the new array because the wrapping changes. Instead, copy the deque from front to back starting at index 0 of the new array. If we start from the 8-element array just shown and use `addLast` to add R, S, and T, the T will be the 11th element and will therefore trigger a resize. After the resize the deque's internal state will be:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
value	**Q**	**P**	**O**	**N**	**M**	**D**	**E**	**F**	**R**	**S**	**T**									

size **11** *front* **0**

As one last example, if we then called `addFirst` on this larger array and added U, V, and W, the deque would be:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
value	**Q**	**P**	**O**	**N**	**M**	**D**	**E**	**F**	**R**	**S**	**T**							**W**	**V**	**U**

size **14** *front* **17**

Part B (ArrayDeque) Implementation Details:

You will write a class named `ArrayDeque<E>` that implements our provided `Deque<E>` interface. It has the following methods; you must write them with exactly the header shown, so it can be called by the provided classes. You **may define additional methods** in your class if you like, but they must be `private`. You will also need to declare any private data fields needed by the class in order to implement this behavior. All specified methods must run in O(1) average runtime unless otherwise specified.

In addition to implementing the provided `Deque<E>`, your class will implement interface `Iterable<E>` so that it can be used in a for-each loop by the client. This is no additional work since your class will already have an `iterator` method.

```
public ArrayDeque()
```

In this constructor you will initialize a newly constructed empty deque. Give it a default array capacity of 10.

```
public void addFirst(E element)
public void addLast(E element)
```

In these methods you should add the given element value to the *front* or *back* of your deque respectively. If the element is `null`, you should throw a `NullPointerException`. (You may need to resize your array to fit the new element. Resizing is O(*N*) but if you resize to a multiple of the old size, the average runtime for adding is O(1).)

```
public void clear()
```

In this method you should remove all elements from your deque. Make sure that your internal array stores only `null` values after a call to `clear` so that the memory previously occupied by the elements can be freed. This method is O(*N*).

```
public boolean isEmpty()
```

In this method you should return whether your deque does not contain any elements.

```
public Iterator<E> iterator()
```

In this method you should return an iterator that provides access to the elements of your deque in front-to back order. The iterator must support the `hasNext` and `next` operations. You can throw an `UnsupportedOperationException` if your iterator's `remove` method is called. You may assume that the deque is not modified during iteration. (Implement your iterator as a private inner class as shown in lecture. See the Java API docs for more info on `Iterator` methods.)

```
public E peekFirst()
public E peekLast()
```

In these methods you should return the first or last element of your deque respectively without modifying the state of the deque. If the deque does not contain any elements, you should throw a `NoSuchElementException`.

```
public E removeFirst()
public E removeLast()
```

In these methods you should remove and return the first or last element of your deque respectively. If the deque does not contain any elements, you should throw a `NoSuchElementException`. You should `null` out the now-empty array slot.

```
public int size()
```

In this method you should return the number of elements in your deque.

```
public String toString()
```

In this method you should return a string representation of your deque's elements in front-to-back order. This is not the same as the string representation of your array: the array is unfilled, so some indexes might not be shown; and the data in the deque may not start at index 0 of the array. For example, if the deque contains D, E, and F at array indexes 3-5 like in the second picture on the previous page, you would return `"[D, E, F]"`. For empty deque you should return `"[]"` and for a one-element deque you should return just that one element in brackets such as `"[hello]"`. This method is O(*N*).

Notice that unlike in past assignments, you may not assume that parameters passed to you are valid and must now throw specific exceptions when bad parameters are passed to you.

Development Strategy and Hints:

This assignment (especially Part B) is tricky, but you can work at it one step at a time. Here is a suggested plan of attack:

- Write Part A first to get to know deques and how they work. Our `MazeSolver` file is 50 lines with comments.
- Implement a basic deque that supports `addLast` and `removeLast`. This is very similar to an array stack.
  At this point you can also write `isEmpty` and `size`.
- Write a temporary incorrect version of `toString` that just returns the `Arrays.toString` of your inner array.
  This is not the proper behavior, but it will help you debug in the meantime.
- Write the `peek` methods.
- Write `removeFirst` so that the front element can be removed and returned without shifting the array elements.
  Make sure that your previously existing methods such as `addLast` and `size` still work properly.
  Don't worry about wrap-around issues or full-array resizing yet.
- Make the array wrap around and implement `addFirst` and patching `addLast`, `removeFirst`, `removeLast`.
  This is hard. Use lots of `println` statements to see the state of your array and other data fields.
  It can be helpful to think about before/after state and what index is considered the front and back after each add.
  Don't forget that you can print an array by writing: `System.out.println(Arrays.toString(myArray));`
- Make it so that your array can resize to a larger array once it becomes full.
- Write the inner iterator class and the `iterator` method, and a proper version of `toString`, etc.

Style Guidelines and Grading:

We will grade your program's behavior and output to give you an **External Correctness** score. You can use the course web site's Output Comparison Tool to check your maze solver and check test case output for your array deque.

We will also grade your program's code quality (**Internal Correctness**). There are many general standards of Java coding style that you should follow, such as naming, indentation, comments, avoiding redundancy, etc. For a list of these standards, please see the **Style Guide** document posted on the class web site. You should follow all of its guidelines as appropriate on this and future assignments.

In past assignments we have used collections heavily to solve problems. But in this assignment (Part B), you are implementing a collection of your own. **For this reason, you may not use any of Java's built-in data structures (such as the ones from `java.util`) to implement your `ArrayDeque`.** For example, declaring a private field in your class of type `ArrayList` or `Map` would be unacceptable. A solution that disobeys this restriction will receive a very substantial deduction. Your deque's inner array should be the only data structure of any kind that is declared by your code.

Your deque must be implemented as a **circular array buffer** as described in this document to receive substantial credit. Solutions that do not do so will receive substantial deductions for both external and internal correctness. You must implement the wrap-around behavior and resizing behavior properly to receive full credit.

You must also match the expected **Big-Oh** demands of each method. Relaxing the Big-Oh could make it easier to implement the functionality, such as if you were to shift elements or copy the array's entire state frequently or other such things; but the whole point of implementing this structure is to do it efficiently. So this will be a grading focus.

Watch out for **redundant code**. If you are performing identical or very similar commands repeatedly, factor out the common code and logic into a helper method, loop, or other facility to remove the redundancy.

We strongly recommend that you write some **private helper methods** to help you implement the required public functionality of `ArrayDeque`. Think about common operations you would want to perform or values you would want to compute and how methods might be useful in such cases. Our own sample solution uses six small private helpers.

Be mindful of your fields. `MazeSolver` probably should have zero fields. `ArrayDeque` should probably have 3-4 fields at the most. Do not declare a value as a private field unless it is necessary to retain it as part of the state of your object.

Now that we are writing methods that throw exceptions, in your method comment headers make sure to comment what exceptions if any are thrown by each method and under what conditions they are thrown. In addition to including comment headers on each class and each method, also make sure to include inline comments next to any complex or tricky code (for example, in the code for the Part A maze solving algorithm).