Important

There are general homework guidelines you must always follow. If you fail to follow any of the following guidelines you risk receiving a **0** for the entire assignment.

1. All submitted code must compile under **JDK 8**. This includes unused code, so don't submit extra files that don't compile. Any compile errors will result in a 0.
2. Do not include any package declarations in your classes.
3. Do not change any existing class headers, constructors, instance/global variables, or method signatures.
4. Do not add additional public methods.
5. Do not use anything that would trivialize the assignment. (e.g. don't import/use `java.util.ArrayList` for an Array List assignment. Ask if you are unsure.)
6. Always be very conscious of efficiency. Even if your method is to be $O(n)$, traversing the structure multiple times is considered inefficient unless that is absolutely required (and that case is extremely rare).
7. You must submit your source code, the `.java` files, not the compiled `.class` files.
8. After you submit your files, redownload them and run them to make sure they are what you intended to submit. You are responsible if you submit the wrong files.

Deques

In class, you've learned in depth about Stacks and Queues, which are LIFO and FIFO data structures respectively. A Stack required you to add and remove from the same end of the data structure while a Queue required you to add and remove from opposite ends of the data structure.

For this assignment, you will be coding two implementations of a Deque. Deque is short for "double-ended queue", and it's a data structure that allows for efficient adding and removing from both ends of the data structure.

The first implementation is backed by a non-circular Doubly-Linked List, and the second implementation is backed by a circular array.

LinkedDeque

As you will recall from when we learned about LinkedLists, in order to efficiently add/remove from both the front and back in $O(1)$ time, we need a Doubly-Linked List (DLL) with a head and tail pointer. Thus, you will be implementing 4 methods for a DLL which are described in the javadocs of `LinkedDeque.java`.

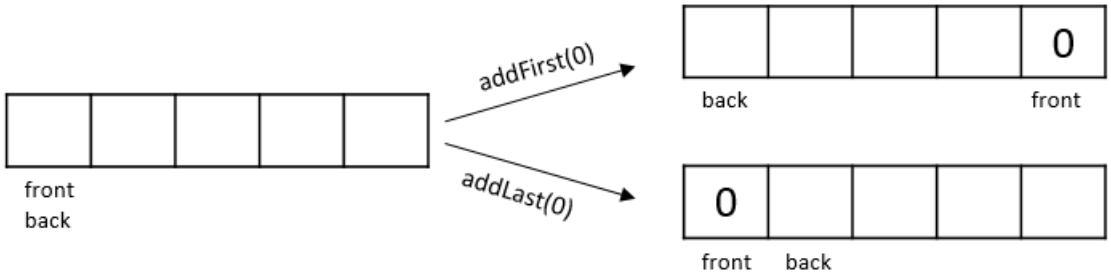
ArrayDeque

The backing array in your `ArrayDeque` implementation must behave circularly. By this, we mean that your front and back variables should wrap around the beginning and end of the array as you add and remove to maintain $O(1)$ efficiency while taking advantage of all the empty space in the array. There is a helper method to help you with the math part of this circular logic near the bottom of the `ArrayDeque.java` file called `mod(int index, int modulo)`, so be sure to look at that as well.

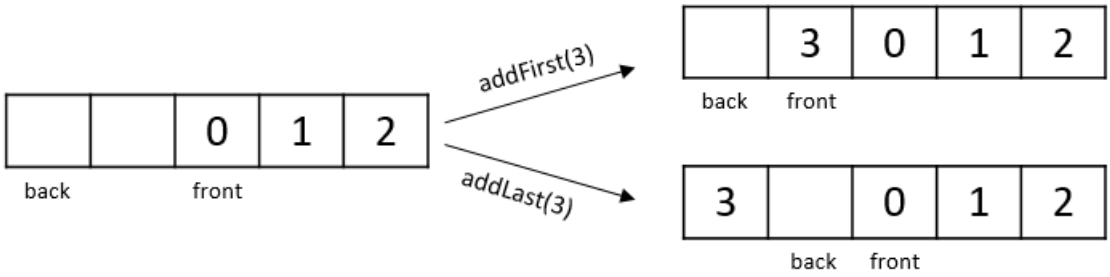
In the `ArrayDeque.java` file, there is an initial capacity constant to use. Do not hardcode the initial capacity; use the given constant instead.

IMPORTANT: Your front variable should represent the index holding the first element of the deque. Your back variable should represent the next index to add to when at the back (usually one more than the index of the last element). **Failure to follow this convention will result in loss of major points.** The examples below demonstrate what the deque should look like at various states.

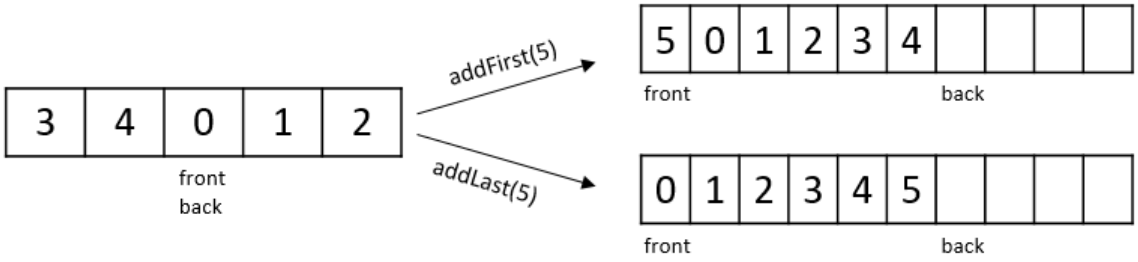
In the example below, the deque begins empty (initial state). When adding to the front here, the front wraps around to the back. The element ends up at the end of the array since adding to the front involves adding to the index before the front variable, which wraps back around.



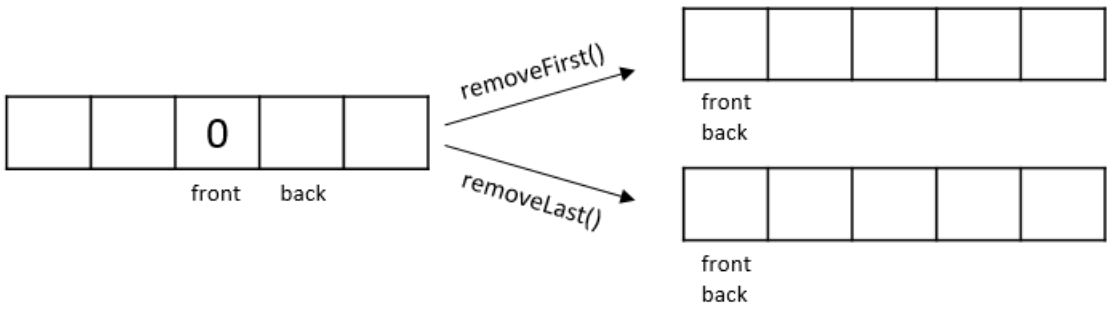
In the example below, adding to the back causes the newly added element to wrap around to the front.



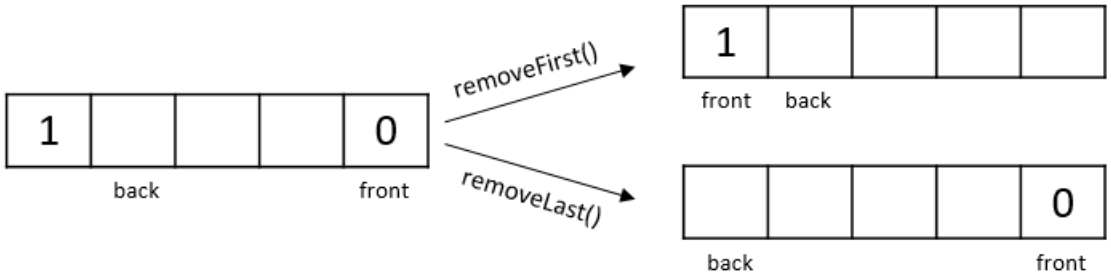
In the example below, adding another element causes the deque to resize since no more space is available, so the array capacity is doubled, the front element moves to index 0 and the back element moves to index `size - 1`. Also note that the front and back variables have moved appropriately to accomodate the change.



In the example below, the last element of the deque is removed, so the front/back are reset to index 0.



In the example below, the front/back wraps around depending on which end is removed from.



Grading

Here is the grading breakdown for the assignment. There are various deductions not listed that are incurred when breaking the rules listed in this PDF, and in other various circumstances.

Methods:	
LinkedDeque addFirst	8pts
LinkedDeque addLast	7pts
LinkedDeque removeFirst	7pts
LinkedDeque removeLast	8pts
ArrayDeque addFirst	13pts
ArrayDeque addLast	13pts
ArrayDeque removeFirst	9pts
ArrayDeque removeLast	10pts
Other:	
Checkstyle	10pts
Efficiency	15pts
Total:	100pts

Keep in mind that add functions are necessary to test other functions, so if an add doesn't work, remove tests might fail as the items to be removed were not added correctly. Additionally, the size function is used many times throughout the tests, so if the size isn't updated correctly or the method itself doesn't work, many tests can fail.

A note on JUnits

We have provided a **very basic** set of tests for your code, in `DequeStudentTests.java`. These tests do not guarantee the correctness of your code (by any measure), nor does it guarantee you any grade. You may additionally post your own set of tests for others to use on the Georgia Tech GitHub as a gist. Do **NOT** post your tests on the public GitHub. There will be a link to the Georgia Tech GitHub as well as a list of JUnits other students have posted on the class Piazza.

If you need help on running JUnits, there is a guide, available on Canvas under Files, to help you run JUnits on the command line or in IntelliJ.

Style and Formatting

It is important that your code is not only functional but is also written clearly and with good style. We will be checking your code against a style checker that we are providing. It is located on Canvas, under Files, along with instructions on how to use it. We will take off a point for every style error that occurs. If you feel like what you wrote is in accordance with good style but still sets off the style checker please

email Tim Aveni (tja@gatech.edu) with the subject header of "[CS 1332] CheckStyle XML".

Javadocs

Javadoc any helper methods you create in a style similar to the existing Javadocs. If a method is overridden or implemented from a superclass or an interface, you may use `@Override` instead of writing Javadocs. Any Javadocs you write must be useful and describe the contract, parameters, and return value of the method; random or useless javadocs added only to appease Checkstyle will lose points.

Vulgar/Obscene Language

Any submission that contains profanity, vulgar, or obscene language will receive an automatic zero on the assignment. This policy applies not only to comments/javadocs but also things like variable names.

Exceptions

When throwing exceptions, you must include a message by passing in a String as a parameter. **The message must be useful and tell the user what went wrong**. "Error", "BAD THING HAPPENED", and "fail" are not good messages. The name of the exception itself is not a good message.

For example:

Bad: `throw new IndexOutOfBoundsException("Index is out of bounds.");`

Good: `throw new IllegalArgumentException("Cannot insert null data into data structure.");`

Generics

If available, use the generic type of the class; do **not** use the raw type of the class. For example, use `new LinkedNode<Integer>()` instead of `new LinkedNode()`. Using the raw type of the class will result in a penalty.

Forbidden Statements

You may not use these in your code at any time in CS 1332.

- `package`
- `System.arraycopy()`
- `clone()`
- `assert()`
- `Arrays` class
- `Array` class
- `Thread` class
- `Collections` class
- `Collection.toArray()`
- Reflection APIs
- Inner or nested classes

- Lambda Expressions
- Method References (using the :: operator to obtain a reference to a method)

If you're not sure on whether you can use something, and it's not mentioned here or anywhere else in the homework files, just ask.

Debug print statements are fine, but nothing should be printed when we run your code. We expect clean runs - printing to the console when we're grading will result in a penalty. If you submit these, we will take off points.

Provided

The following file(s) have been provided to you. There are several, but we've noted the ones to edit.

1. `LinkedDeque.java`

   This is the class in which you will implement the linked list-backed deque. Feel free to add private helper methods but **do not add any new public methods, inner/nested classes, instance variables, or static variables**.

2. `ArrayDeque.java`

   This is the class in which you will implement the array-backed deque. Feel free to add private helper methods but **do not add any new public methods, inner/nested classes, instance variables, or static variables**.

3. `LinkedNode.java`

   This class represents a single node in the linked list. It encapsulates `data`, `previous`, and `next` references. **Do not alter this file.**

4. `DequeStudentTests.java`

   This is the test class that contains a set of tests covering the basic operations of your implementations. It is not intended to be exhaustive and does not guarantee any type of grade. **Write your own tests to ensure you cover all edge cases.**

Deliverables

You must submit **all** of the following file(s). Please make sure the filename matches the filename(s) below, and that *only* the following file(s) are present. If you make resubmit, make sure only one copy of the file is present in the submission.

After submitting, double check to make sure it has been submitted on Canvas and then download your uploaded files to a new folder, copy over the support files, recompile, and run. It is your responsibility to re-test your submission and discover editing oddities, upload issues, etc.

1. `LinkedDeque.java`
2. `ArrayDeque.java`