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 11**. 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. For example, do not add `throws` to the method headers since they are not necessary.
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 `ArrayList` assignment. Ask if you are unsure.)
6. You are expected to implement all of the methods in this homework.
7. You must submit your source code, the `.java` files, not the compiled `.class` files.
8. Only the last submission will be graded. Make sure your last submission has **all** required files. Resubmitting will void all previous submissions.
9. 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.

Collaboration Policy

Every student is expected to read, understand and abide by the Georgia Tech Academic Honor Code.

When working on homework assignments, you **may not** directly copy code from any source (other than your own past submissions). You are welcome to collaborate with peers and consult external resources, but you **must** personally write all of the code you submit. **You must list, at the top of each file in your submission, every student with whom you collaborated and every resource you consulted while completing the assignment**.

You may not directly share any files containing assignment code with other students or post your code publicly online. If you wish to store your code online in a personal private repository, you can use Github Enterprise to do this for free.

The only code you may share is JUnit test code on a pinned post on the official course Piazza. Use JUnits from other students at your own risk; **we do not endorse them**. See each assignment's PDF for more details. If you share JUnits, they **must** be shared on the site specified in the Piazza post, and not anywhere else (including a personal GitHub account).

Violators of the collaboration policy for this course will be turned into the Office of Student Integrity.

Style and Formatting

It is important that your code is not only functional, but written clearly and with good programming style. Your code will be checked against a style checker. The style checker is provided to you, and is located on Canvas. It can be found under Files, along with instructions on how to use it. A point is deducted for every style error that occurs. If there is a discrepancy between what you wrote in accordance with good style and the style checker, then address your concerns with the Head TA.

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.

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 LinkedList<Integer>()` instead of `new LinkedList()`.

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.

JUnits

We have provided a **very basic** set of tests for your code. These tests do not guarantee the correctness of your code (by any measure), nor do they 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.

CircularSinglyLinkedList

You are to code a `CircularSinglyLinkedList` with a head reference. A linked list is a collection of nodes, each having a data item and references to other nodes. In a `CircularSinglyLinkedList`, each node has a reference to the next node. Since it must be circular, the next reference for the last node in this list will point to the head node. As a special case, this means that in a one node list, the head node will point to itself.

Do **not** use a phantom node to represent the start or end of your list. A phantom or sentinel node is a node that does not store data held by the list and is used solely to indicate the start or end of a linked list. If your list contains n elements, then it should contain exactly n nodes.

The `CircularSinglyLinkedList` must follow the requirements stated in the javadocs of each method you must implement. Your linked list implementation will use the default constructor (the one with no parameters) which is automatically provided by Java. Do **not** write your own constructor.

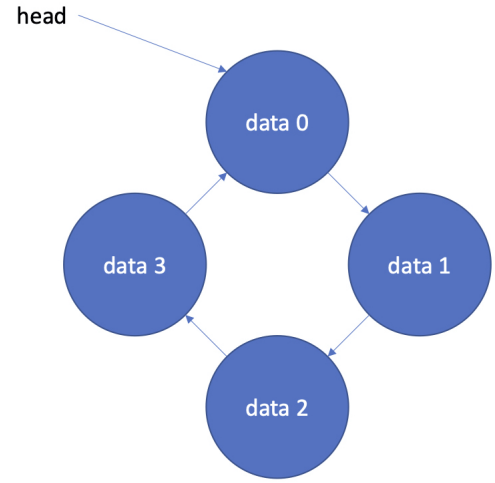
As an additional note, your circular implementation doesn't have a tail reference, but it is still possible to efficiently add and remove from the head as well as add to the back in $O(1)$ time. However, it is still not possible to remove from the back in $O(1)$ time unless the linked list is doubly-linked.

The examples below demonstrate what the `CircularSinglyLinkedList` should look like at various states.

Size 0:

Size 1:

Size 4:



Nodes

The linked list consists of nodes. A class `CircularSinglyLinkedListNode` is provided to you. This class has getter and setter methods to access and mutate the structure of the nodes.

Adding

You will implement three `add()` methods. One will add to the front, one will add to the back, and one will add to anywhere in the list given a specific index. See the javadocs for more details.

Removing

You will also implement three `remove()` methods - from the front, the back, or anywhere in the list given a specific index. Make sure that there is no longer any way to access the removed node so that the node will be garbage collected. See the javadocs for more details.

Garbage Collection

Java will automatically mark objects for garbage collection based on whether there is any means of accessing the object. In other words, if we want to remove a node from the list, we must remove all references **to that node**. What the next reference of that node points to doesn't particularly matter. As long as no references can reach the node, the node will be garbage collected eventually.

Equality

There are two ways of defining objects as equal: reference equality and value equality.

Reference equality is used when using the `==` operator. If two objects are equal by reference equality, that means that they have the exact same memory locations. For example, say we have a Person object with a name and id field. If you're using reference equality, two Person objects won't be considered equal unless they have the exact same memory location (are the exact same object), even if they have the same name and id.

Value equality is used when using the `.equals()` method. Here, the definition of equality is custom made for the object. For example, in that Person example above, we may want two objects to be considered equal if they have the same name and id.

Keep in mind which makes more sense to use while you are coding. **You will want to use value equality in most cases in this course when comparing objects. Notable cases where you'd use reference equality include checking for null or comparing primitives (in this case, it's just the == operator being overloaded).**

Differences between Java API and This Assignment

Some of the methods in this assignment are called different things or don't exist in Java's `LinkedList` class. Additionally, Java's built in `LinkedList` is a Doubly-Linked List, so the efficiency of some operations will differ. This won't matter until you tackle coding questions on the first exam, but it's something to be aware of. The list below shows all methods with a different name and their Java API equivalent if it exists. The format is assignment method name ⇒ Java API name.

- `addAtIndex(int index, T data)` ⇒ `add(int index, T data)`
- `addToFront(T data)` ⇒ `addFirst(T data)`
- `addToBack(T data)` ⇒ `add(T data)` or `addLast(T data)`

- `removeAtIndex(int index)` ⇒ `remove(int index)`
- `removeFromFront()` ⇒ `poll()` or `pollFirst()`
- `removeFromBack()` ⇒ `pollLast()`

Grading

Here is the grading breakdown for the assignment.

Methods:	
addAtIndex	10pts
addToFront	5pts
addToBack	5pts
removeAtIndex	10pts
removeFromFront	5pts
removeFromBack	5pts
get	10pts
isEmpty	4pts
clear	5pts
removeLastOccurrence	10pts
toArray	6pts
Other:	
Checkstyle	10pts
Compiling	15pts
Total:	100pts

Provided

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

1. `CircularSinglyLinkedList.java`

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

2. `CircularSinglyLinkedListNode.java`

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

3. `CircularSinglyLinkedListStudentTest.java`

   This is the test class that contains a set of tests covering the basic operations on the `CircularSinglyLinkedList` class. 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). Make sure all file(s) listed below are in each submission, as only the last submission will be graded. Make sure the filename(s) matches the filename(s) below, and that *only* the following file(s) are present. If there are multiple files, do not zip up the files before submitting; submit them all as separate files.

Once submitted, double check that it has uploaded properly on Gradescope. To do this, download your uploaded file(s) to a new folder, copy over the support file(s), recompile, and run. It is your sole responsibility to re-test your submission and discover editing oddities, upload issues, etc.

1. `CircularSinglyLinkedList.java`