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 `ArrayList` 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.

ArrayList

You are to code an `ArrayList`, which is a list data structure backed by an array where all of the data is contiguous and aligned with index 0 of the array.

The `ArrayList` must follow the requirements stated in the javadocs of each method you must implement. A constructor stub is provided for you to fill out. **Do not** change headers for the constructor and for any of the methods provided. For example, many of the methods require you to throw exceptions, but you should not add "throws" to the header since they are not necessary.

Capacity

The starting capacity of the `ArrayList` should be the constant `INITIAL_CAPACITY` defined in `ArrayList.java`. Reference the constant as-is. **Do not** simply copy the value of the constant. **Do not** change the constant. If, while adding an element, the `ArrayList` does not have enough space, you should regrow the backing array to **twice** its old capacity. **Do not** regrow the backing array when removing elements.

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. When adding to the front or the middle of the list, subsequent elements must be shifted back one position to make room for the new data. See the javadocs for more details.

Removing

Just like add, you will implement three `remove()` methods - from the front, the back, or anywhere in the list. When removing from the front or from the middle of the list, the element should be removed and all subsequent elements should be shifted forward by one position. When removing from the back, the last element should be set to `null` in the array. **All unused positions in the backing array must be set to `null`.** See the javadocs for more details.

Amortized Efficiency

The efficiency of methods and algorithms in this course is often analyzed using a "per operation" analysis. That is, what is the worst this algorithm can do on any one instance? However, there are times where this type of analysis is unrealistically pessimistic. For example, in this homework, the `addToBack()` method is $O(1)$ for the most part except in the case of resizing, which is $O(n)$. However, a resize operation is rare enough that it'd be misleading to say that the method is $O(n)$.

In cases like this, we use an **amortized analysis**. This type of analysis adds up the cost of a series of operations and then averages the cost. Here, the resize step is $O(n)$, but since we double the capacity whenever the array gets full, we've put off resizing for another n add operations. So, putting that together with the common, cheap $O(1)$ operations, we get $O(1)$ using this analysis. Whenever this type of analysis is used, we will prefix the Big-O with the word **amortized**.

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 `ArrayList` class. 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)` ⇒ no explicit method
- `addToBack(T data)` ⇒ `add(T data)`
- `removeAtIndex(int index)` ⇒ `remove(int index)`
- `removeFromFront()` ⇒ no explicit method
- `removeFromBack()` ⇒ no explicit method

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:	
addAtIndex	15pts
addToFront	11pts
addToBack	11pts
removeAtIndex	11pts
removeFromFront	7pts
removeFromBack	7pts
get	6pts
isEmpty	3pts
clear	4pts
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.

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.

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 Adrianna Brown (adrianna.brown@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 LinkedList<Integer>()` instead of `new LinkedList()`. 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. `ArrayList.java`

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

2. `ArrayListStudentTest.java`

   This is the test class that contains a set of tests covering the basic operations on the `ArrayList` 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). Please make sure the filename(s) 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. `ArrayList.java`