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, 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 a 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.

Singly-Linked List

You are to code a singly-linked list with a head and a tail reference. A linked list is a collection of nodes, each having a data item and a reference pointing to the next node. The next reference for the last node in this list must be `null`, as it is not circular. 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.

Your linked list implementation will implement the `LinkedListInterface` provided. It will use the default constructor (the one with no parameter) which is automatically provided by Java. Do not write your own constructor.

Nodes

The linked list consists of nodes. A class `SLLNode` is provided to you. `SLLNode` has three methods to access and set the next node, and to access the data in the node.

Adding

You will implement three `add()` methods. One will add to the front, one will add to the back, and one will add anywhere in the list. When adding the first element to an empty list, the new node should be both the head and the tail. See the interface for more details.

Removing

Removing, just like adding, can be done from the front, the back, or anywhere in your linked list. In addition, the first instance of a specific value from the list can be removed. When removing from the front, the first node should be removed and the head reference should be updated. When removing from the back, the last node should be removed and the tail reference should be updated. When removing

from the middle, the previous node of the removed node should point to the next node of the removed node. Make sure that you set any pointers to the deleted nodes to `null`. See the interface for more details.

Comparaing Data

For the `findLargestElement()` method, you will need to compare the elements in your list to the given data. You may be tempted to use `==` like with primitives, but this does not work as expected with objects. In Java, using the `==` operator, `<` operator, `>` operator, `<=` operator, or `>=` operator are valid for comparing primitive types, but are not valid for comparing objects. Java will not allow you to use binary operators to compare objects. Therefore, the generic type for `SinglyLinkedList` implements the `Comparable` interface, and you can use the `compareTo()` method to compare two objects.

Recursion

You are to code two methods whose solutions must be written recursively. Recursion is a means of breaking down a problem into a set of smaller sub-problems. This process of breaking down the problem ends when we reach a "base case", a condition to which a problem is at its most basic form and can be solved without recursion. In regards to functions, recursion is when a function calls itself but in a modified and simpler state. The calls will eventually converge to the base case, in which the recursive call will return through the previous calls, utilizing the solution of current call as part of the solution to the caller. Your solution must be in what is considered "proper format": a base case check followed by logic that will reduce the recursion toward the base case.

Pseudocode:

```
recursiveMethod(x):
    if (x == 0) {
        print "We reached the base case!"
    } else {
        print "Let's get closer"
        recursiveMethod(x-1)
    }
```

Private recursive helper methods

Note that you **must** utilize a private helper method to execute the logic of your recursion. The main reason for this is that it adds a layer of abstraction between the main method and the recursive logic. In most cases, the recursion requires the use of more parameters than the main call has in its method signature. Therefore, we will make a private method that has the additional parameters that we need. Both methods in `Recursion.java` must have private helper methods. All private helper methods, as per Checkstyle requirements, **must** have proper Javadocs.

Palindromes

A palindrome is a sequence of characters such that it can be read forwards and backwards with the same resulting test. An example of a palindrome is "A nut for a jar of tuna". Logically, we can see that if we remove the 'a' at the front and at the back, the remaining text is still a palindrome. This leads us to the recursive definition of a palindrome is another palindrome with a character appended to the front and the same character appended to the back. We will consider a sequence of one or zero characters being a palindrome.

Greatest Common Divisor (GCD)

The greatest common divisor is the largest number that can divide two or more numbers evenly. Euclid's algorithm is a recursive solution to calculate the GCD of two numbers. More details about GCD are provided in the Javadocs in the `Recursion.java` file.

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	9pts
addToFront	3pts
addToBack	3pts
removeAtIndex	8pts
removeFromFront	4pts
removeFromBack	5pts
findLargestElement	8pts
get	8pts
toArray	4pts
clear	5pts
isEmpty	4pts
gcd	7pts
isPalindrome	7pts
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 `SinglyLinkedList.java` and `Recursion.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 (when it comes up).

If you need help on running JUnits, there is a guide, available on T-Square under Resources, 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 in T-Square, under Resources, 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 Grayson Bianco (gbianco6@gatech.edu) with the subject header of "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.

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:

```
throw new PDFReadException("Did not read PDF, will lose points.");

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

- `break` may only be used in switch-case statements
- `continue`
- `package`
- `System.arraycopy()`
- `clone()`
- `assert()`
- `Arrays` class
- `Array` class
- `Collections` class
- `Collection.toArray()`
- Reflection APIs
- Inner or nested classes

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 you will only edit one of them.

1. `LinkedListInterface.java`

   This is an interface you will implement. All instructions for what the methods should do are in the javadocs. **Do not alter this file.**

2. `SinglyLinkedList.java`

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

3. `SLLNode.java`

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

4. `SinglyLinkedListStudentTests.java`

   This is the test class that contains a set of tests covering the basic operations on the `SinglyLinkedList` 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.**

5. `Recursion.java`

   This is a class in which you will implement the recursive method solutions.All instructions for what the methods should do are in the javadocs.

6. `RecursionStudentTests.java`

   This is the test class that contains a set of tests covering the basic operations on the `Recursion` 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 matches the filename(s) below. Be sure you receive the confirmation email from T-Square, and then download your uploaded files to a new folder, copy over the interfaces, recompile, and run. It is your responsibility to re-test your submission and discover editing oddities, upload issues, etc.

1. `SinglyLinkedList.java`
2. `Recursion.java`