Homework 1: Lists, Stacks, Queues

CS 1332 Section C

Fall 2021

1 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. You may create helper methods, but any helper method you create should be **private**.
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 consider the efficiency of your code. Even if your method is *O(n)*, traversing the structure multiple times is considered inefficient unless that is absolutely required (and that case is very rare).
7. You must submit your source code - the .java files. Do not submit compiled code - the .class files.
8. Only the last submission will be graded. Make sure your last submission has all required files. Resubmitting voids prior submissions.

2 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 it located on Canvas. A point is deducted for every style error that occurs. Please double check before you submit that your code is in the appropriate style so that you don't lose any unnecessary points!

2.1 Javadocs

Javadocs should be written for any private helper methods that you create. They should follow a style similar to the existing javadocs on the assignment. 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.

2.2 Vulgar/Obscene Language

Any submission that contains profanity, vulgar, or obscene language will receive an automatic zero on the assignment. This policy applies to all aspects of your code, such as comments, variable names, and javadocs.

2.3 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”, ”Oof - Bad things are happening”, and ”FAIL” are *not* good messages. Additionally, the name of an exception itself is not a good message.

In addition, you may not use try catch blocks to catch an excpetion unless you are catching an exception you have explicitly thrown yourself with the **throw new ExceptionName(’Exception Message”);** syntax

2.4 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.

3 Forbidden Statements

You may not use any of the following in your code at any time in CS 1332. If you are not sure whether you can use something, and it is not explicitly listed here, just ask. Debug print statements are fine, but should be either removed or commented out prior to submission. If print statements are left in, assignments are messy to grade, and checkstyle points will be deducted.

- package
- System.arraycopy()
- clone()
- assert()
- Arrays class
- Thread class
- Collections class
- Collection.toArray()
- Refleciton APIs
- Inner or nested classes
- Lambda Expressions
- Method References (using the :: operator to obtain a reference to a method)

4 JUnits

We have provided a **very basic** set of tests for your code. These tests do not guarantee the correctness of your code, 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.

5 Deliverables

You must submit all of the following file(s) to the corresponding assignment on Gradescope. Make sure all file(s) listed below are in each submission, as only the last submission is graded. Make sure the filename(s) matches the filename(s) below, and that *only* the following file(s) are present.

1. ArrayList.java
2. ArrayQueue.java
3. LinkedStack.java
4. CircularSinglyLinkedList.java

5.1 Array List

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 the 0 index 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.

5.1.1 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 resize the backing array when removing elements.

5.1.2 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 as specific index. 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.

5.1.3 Removing

You will also implement three remove() methods - from the front, back, or anywhere in the list given a specific index. When removing from the front or anywhere in 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.

5.1.4 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 **amortized analysis**. This type of analysis looks at the algorithm as a whole and excludes the rare, infrequent edge cases (which in this case is the resize). Here, the resize step is O(n), but since we double the capacity only whenever the array gets full, we've put off resizing for another n add operations. So, the larger our array becomes, the more common the O(1) case occurs and the less common the O(n) resize occurs. We call this "O(1) amortized" which is sometimes notated as O(1)*.

5.2 Array Queue

A queue is a first-in, first-out (FIFO) data structure; the first item inserted is the first item to be removed. For this assignment, you will create a **queue** backed by an array.

5.2.1 Circular Arrays

The backing array in your **ArrayQueue** implementation must behave circularly. This means the front variable might wraparound to the beginning when you remove to take advantage of empty space while maintaining O(1) efficiency for all operations.

For this assignment, the front variable should represent the index that holds the next element to dequeue. **Failure to follow this convention will result in major loss of points**.

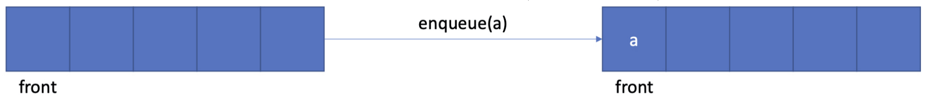
For enqueuing, add to the back of the queue. To access the back of the queue, you can add the size to the front variable to get the next index to add to, though you will have to account for the circular behavior yourself. If there are empty spaces at the front of the array, the back of the queue should wrap around to the front of the array and make use of those spaces.

For dequeuing, you should simply treat the next index in the array as the new front. Do not shift any elements during a remove. Additionally, after removing the last element in the queue, do **NOT** explicitly reset the front to index 0. This means that going from size 1 to 0 should not be a special case for your code.

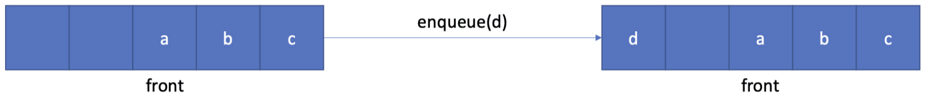
When resizing the backing array, "unwrap" the data. Realign the queue with the front of the new array during the transfer. The front variable of the queue is once again at index 0.

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

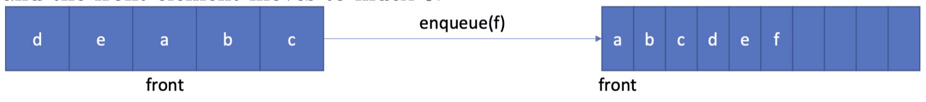
In the example below, the queue begins empty (initial state). An element is added.



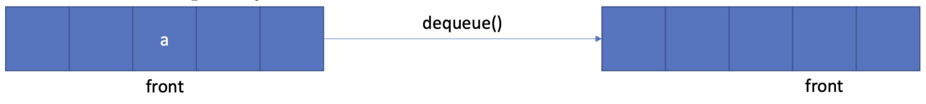
In the example below, adding to the back causes the newly added element to wrap around to the front.



In the example below, adding another element causes the queue to resize. The array capacity is doubled and the front element moves to index 0.



In the example below, the last element of the queue is removed, but front moves as expected. The front variable is not explicitly set to 0.



5.3 Linked Stack

A stack is a last-in, first-out (LIFO) data structure; the last item inserted is the first item to be removed. For this assignment, you will create a **stack** backed by as singly linked list without a tail reference.

Note: You must use the provided LinkedNode class in your implementation. Refer to javadocs for more details.

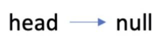
5.4 Circular Singly Linked List

The final part of this assignment will be coding a CircularSinglyLinkedList with 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 references to the next node. Since it must be circular, the next reference for the last node of the list should point to the head node. As a special case, this means that in a one node list, the head node should point to itself.

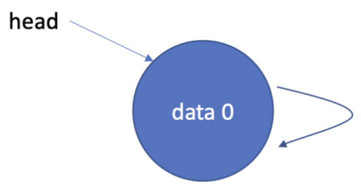
Do not use a phantom node to represent the start or end of your list. A phantom node is a node that does not contain data held by the list and is used solely to indicate the start or end of a linked list. If your list contains n elements, there should be exactly n nodes.

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.

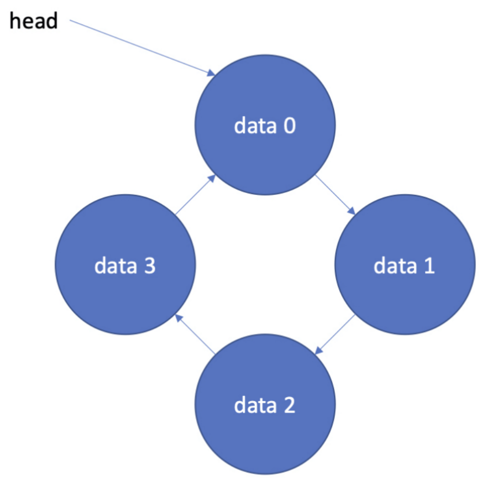
Size 0:



Size 1:



Size 4:



6 Grading

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

- Array List – 30pts
- Array Queue – 15pts
- Linked Stack – 15pts
- Circular Singly Linked List – 30pts
- Checkstyle – 10 pts
- Total: 100 pts