PA 1
Due Date: Wednesday 13 September 2023 by 11:59 PM

General Guidelines.

The instructions given below describe the required methods in each class. You may need to write additional methods or classes that will not be directly tested but may be necessary to complete the assignment.

In general, you are not allowed to import or use any additional classes in your code without explicit permission from your instructor!

Note: It is okay to use the Math class if you want.

Project Overview

In this class, we are very concerned about how data structures are actually implemented, especially in terms of efficiency. In this project, you will implement a dynamic list structure using an array, similar to the ArrayList ADT in Java. There will also be a written portion that asks you to think through some of the runtime details.

Part 0. Before you start coding.

Before you start coding, it is a good idea to make sure you have a current account on lectura and you know your password. You should also go through the process of transferring files to lectura to make sure you know how to do it. The following are resources that may help with this process.

- mac to lectura tutorial
- lectura instructions for pc
- CS Help Desk (in case you need help with account information)
- TAs & instructors (We can walk you through the process in office hours or SI.)

Part 1. ArrList.java

I've found that students who code in Java often use the ArrayList data structure as it is such a flexible structure with so many operations. Although there is nothing wrong with using an ArrayList in general, in this class we are very careful when and how we are allowed to use it. That is because some operations are relatively efficient while some are not, and to be a good programmer, you should really have a good understanding of how data structures are implemented so that you can make good, informed decisions.

An ArrayList is called an ArrayList because it is a list built with an array. That means that it is built as a growable structure from a fixed-size structure (an array).

To better understand how this affects efficiency and how some operations can be implemented in an efficient way, you are going to write your own ArrList data structure. You must use the Array class as your implementation will be tested for efficiency based on how many array accesses you use.

Required Methods for ArrList.java

Method Signature	Description
`ArrList()`	`constructor: default size of Array should be 10`
`ArrList(int cap)`	`constructor: starting Array size is <cap>`
`void addLast(int num)`	`adds a new element to the end of the list//should be O(1) except when resizing is necessary`
`void addFirst(int num)`	`adds a new element to the front of the list//should be O(1) except when resizing is necessary`
`int get(int i)`	`returns the element at index i (of the ArrList)//should be O(1)`
`int indexOf(int num)`	`return the ArrList index of the first occurrence of <num> or -1 if <num> is not in the list//should be O(N)`
`boolean contains(int num)`	`return true if the list contains <num> and false otherwise//should be O(N)`
`isEmpty()`	`return true if the list is empty and false otherwise//should be O(1)`

`int lastIndexOf(int num)`	return the ArrList index of the last occurrence of <num> or -1 if <num> is not in the list//should be O(N)
`int removeFirst() throws EmptyListException`	remove and return the first element in the list; throw the exception if the list is empty//should be O(1) unless resizing is necessary
`int removeLast() throws EmptyListException`	remove and return the last element in the list; throw the exception if the list is empty//should be O(1) unless resizing is necessary
`int removeByIndex(int i) throws EmptyListException`	remove and return the element at index i and close the gap; throw the exception if the list is empty//can be O(N) due to closing the gap but should be O(1) if there is no gap to close (i.e. if <i> is 0 or *list.size()-1*).
`boolean removeByValue(int num) throws EmptyListException`	remove the first occurrence of element <num> from the list if it exists; close the gap; return true if the element was removed and false otherwise; throw the exception if the list is empty//should be O(N)
`int set(int index, int num)`	set the value at <index> to <num>//should be O(1)
`int size()`	return the number of elements in the list//should be O(1)

Guidelines:

- Implement your solution in a class called *ArrList.java*

- You are not allowed to use any additional data structures (besides regular single-value variables)–only the underlying Array (but this will need to be resized).
- Familiarize yourself with the Array class if you haven't already. There are many methods provided already, including a resizing method. You should use the resizing method that is provided.
- Some of the operations above are difficult (if not impossible) to do very efficiently (i.e. inserting or removing from the middle of the list). However, adding/removing from the front or the back of the list can be done with reasonable efficiency if you follow these guidelines:
    - Resize your underlying array by multiplying it by some factor (not by adding a constant amount)--for example, multiply it by 2 when it gets full or cut it in half when it drops below ¼ full. This ensures that you aren't wasting too much space while still providing a reasonably efficient runtime. The justification for this is called *amortized cost analysis* and can be shown mathematically. Also, resize when you are trying to add an element to a full Array–that means you should check to see if the Array needs to be resized at the beginning of any "add" method.
    - When removing an element, you should remove the element first, then check if the Array is less than ¼ full. In that case, resize to half the capacity *unless that would put the capacity below 10.*
    - Some of these methods are required to "throw" an exception. The Exception code is provided for you, but if you are unfamiliar with exceptions in Java, you may want to look up how to use exceptions. Some of you may wonder if the test code is incorrect in how it is handling exceptions. It is not. It is handling the exceptions with try-catch blocks because it is expecting your method to "throw" the exception in the appropriate place.
    - Use a wrap-around method for adding/removing from the front and back of the list.
        - This will require you to keep track of where the *front* and *back* elements of the list are.
        - This will also mean that the index of the ArrList may not be the same as the index for the underlying Array. All indexes in the ArrList API are ArrList indexes. You will need to go back and forth between those in your implementation.
        - Consider the example below and think about where the *front* and *back* of the list are in each case.

```
addLast(5)
```

5						

`addLast(6)`

5	6					

`addLast(7)`

5	6	7				

`addFirst(8)`

5	6	7				8

`addFirst(9)`

5	6	7			9	8

`addLast(10)`

5	6	7	10		9	8

`removeFirst()`

5	6	7	10			8

`removeLast()`

5	6	7			9	8

Part 2. Written Report

Submit these answers to Gradescope as a PDF. Make sure you assign pages to questions appropriately or you will receive a 3-point deduction.

Question 1. Without factoring in the occasional resizing of the array, the project required you to implement *addFirst*, *addLast, removeFirst*, and *removeLast* in constant (O(1)) time using a circular, wrap-around array. Briefly describe how you achieved this and why the runtime is O(1).

Question 2. In order to use the circular array method, the index of the ArrList structure has to be mapped to the actual index in the array. Explain this mapping.

Question 3. The runtime for adding or removing to the middle of the ArrList was not expected to be better than O(N). Explain why you would not be expected to do this in constant time.

Question 4. Look up Stacks, Queues, and Deques. Briefly describe them here. Then, based on what you did in the ArrList.java, explain how Stacks, Queues, and Deques can be implemented efficiently with an array (not factoring in the resizing).

Question 5*. The previous questions do not factor in the resizing of the array for the *addFirst, addLast, removeFirst,* and *removeLast* operations. But it is something to consider because when the resizing is done, the runtime for a single operation becomes O(N), making the worst-case guarantee much worse than O(1). The reason why we can still consider this implementation to be reasonably efficient is due to the resizing strategy and something called *amortized analysis.* Read about Amortized Analysis here and explain in your own words (a) what amortized analysis is and (b) how you might go about doing amortized analysis on these ArrList operations using the array access count.

*This question is graded for completion.

Testing & Submission Procedure.

This time, test code is provided for you, but keep in mind that the test code may not be exhaustive and catch all possible errors. The test code also compares your access counts to the expected ones, so you should get a good idea of whether or not your efficiency is okay. Again, you should not rely solely on the test cases as your code will also be checked manually. Make sure your code runs with the test code *as it is*. Since you won't be submitting the test code file, your submission cannot depend on any changes you make to that file. The same is true for the Array.java file. You can change it for yourself for testing purposes, but we will use the original one when testing, so make sure your code works with that.

Note on Efficiency Testing. Unfortunately, there is no foolproof way to autograde for efficiency. The tests provided are meant to give you a reasonable idea of whether your solutions fits within the expected efficiency level and what is possible. Ultimately, we can always check your code manually if necessary (we do some of this already) to verify that the autograder is returning reasonable results.

Your code must compile and run on lectura, and it is up to you to check this before submitting.
To test your code on lectura:

- Transfer all the files (including test files) to lectura.
- Run *javac *.java* to compile all the java files. (You can also use *javac* with an individual java file to compile just the one file: *javac ArrList.java*)

- Run *java <filename>* to run a specific java file. (Example: *java ArrListTest.java*)

After you are confident that your code compiles and runs properly on lectura, you can submit the required files using the following **turnin** command. Please do not submit any other files than the ones that are listed.

```
turnin csc345pa1 ArrList.java
```

Upon successful submission, you will see this message:

```
Turning in:
      ArrList.java -- ok
All done.
```

Note: Only properly submitted projects are graded! No projects will be accepted by email or in any other way except as described in this handout. If you are worried about your submission, feel free to ask us to check that it got into the right folder.

Grading.

Auto-grading

Part	Total Points	Details
ArrList.java	37	• 11 method tests: 2 points for correctness + 1 point for efficiency • 1 final test: 2 points for correctness + 1 point for efficiency
Coding Style	8	We will check for • understandable code • helpful comments • good indentation • good method abstraction–i.e. you should be writing reusable methods rather than repeating the same code multiple times
Written Report	15	3 points per question; some may be graded for accuracy, some for completion

Your score will be determined by the tests we do on your code minus any deductions that are applied when your code is manually graded. In addition to late deductions, coding style, and deductions for not following directions, you may also receive deductions for inefficient code.

See the late submission and resubmission policies in the syllabus.