Project 2

Due Date: Wednesday 23 February 2022 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 any additional classes in your code without explicit permission from your instructor!

Note on academic dishonesty: Please note that it is considered academic dishonesty to read anyone else's solution code, whether it is another student's code, code from a textbook, or something you found online. You MUST do your own work! It is also considered academic dishonesty to share your code with another student. Anyone who is found to have violated this policy will be subject to consequences according to the syllabus and university policy.

Note on grading and provided tests: The provided tests are to help you as you implement the project and (in the case of auto-graded sections) to give you an idea of the number of points you are likely to receive. Please note that the points indicated when you run these tests locally are not your final grade. Your solution will be graded by the TAs after you submit. Please also note that these test cases are not likely to be exhaustive and find every possible error. Part of programming is learning to test and debug your own code, so if something goes wrong, we can help guide you in the debugging process but we will not debug your code for you.

Project Overview.

The ArrayList class in the java.util library tends to be very popular. However, it is not always a good choice to use provided classes and methods without at least a rudimentary understanding of what is likely happening underneath. In this project, you will implement a paired-down version of an ArrayList. Like the ArrayList class in java, you will implement this as a

dynamically resizable array. Hopefully, you will see that some of these methods can be implemented with reasonable efficiency, while others are difficult (if not impossible) to implement efficiently. This does not mean that the methods can never be used, but it does mean that a programmer will likely write more efficient programs if they have at least a mental model of which operations are most efficient and are disciplined about the choices they make.

Learning Goals

- Practice with generics in Java.
- Gain an understanding of how various operations in a dynamically resizable array are likely implemented.
- Understand the runtimes of different operations in a dynamically resizable array.
- Learn some techniques for making operations on an array more efficient.

Required Classes and Methods

ArrList<T>.java: a generic ArrList structure that is implemented with an array and has many of the features of an ArrayList

Required Method Signature	**Description**	**Runtime Requirements: N is the number of elements in the list**
`ArrList()`	`constructor: construct an ArrList with a starting capacity of 10`	
`ArrList(int cap)`	`constructor: construct an ArrList with a starting capacity of` *`cap`*	

`void add(T t)`	• `if the array is full, resize it to be twice as big`
	• `add item` *`t`* `to the end of`
		`best: O(1)`
		`worst: O(N)`
		`amortized: O(1)`

	the ArrList	
void add(int i, T t)	• add *t* to the list at index *i* • make sure to shift things over as necessary	best: O(1) worst: O(N)
void clear()	• remove all the elements from the ArrList • the final capacity of the array should be 10	best/worst: O(N)
T get(int i)	return the element at index *i*	best/worst: O(1)
int indexOf(Object o)	return the index of the first occurrence of Object o or -1 if o is not in the list	best: O(1) worst: O(N)
boolean contains(Object o)	return true if the list contains o, false otherwise	best: O(1) worst: O(N)
boolean isEmpty()	return true if the list is empty and false otherwise	best/worst: O(1)
int lastIndexOf(Object o)	return the index of the last occurrence of o in the list or -1 if o is not in the list	best: O(1) worst: O(N)
T remove(int i)	• remove the element at index *i* • don't forget to close the gap! • if the number of elements	best: O(1) worst: O(N)

	falls below floor(M/4) (where M is the array's capacity), resize the array to be of size max(floor(M/2), 10) • return the item that was removed	
boolean remove(Object o)	• remove the first occurrence of o from the list if it exists • don't forget to close the gap! • if the number of elements falls below floor(M/4) (where M is the array's capacity), resize the array to be of size max(floor(M/2), 10) • return true if the element was removed and false otherwise	best: O(1) worst: O(N)
void removeRange(int from, int to)	• remove all the elements from index *from* (inclusive) to index *to* (exclusive) • don't forget to close the gap! • if the number of elements falls below floor(M/4) (where M is the array's	best: O(1) worst: O(N)

	capacity), resize the array to be of size max(floor(M/2), 10)	
T set(int index, T item)	• replace the element at index *index* with *item* • return the item that was previously at that index	best/worst: O(1)
int size()	return the number of elements in the list	best/worst: O(1)
void trimToSize()	resize the array so that its capacity is equal to the number of elements in the list	best/worst: O(N)

About Amortized Runtime

Amortized cost analysis is sometimes used when we are talking about operations on data structures. It is especially useful to think about when you have an operation that has a really good best case runtime and a pretty bad worst case runtime but when the **worst case only happens once in a while.** When that happens, it's possible that the worst case will happen infrequently enough (and in a predictable way) that the average cost for that operation is actually quite good.
The amortized cost of an operation is calculated as follows:

- Add up the total cost of doing *M* of the operations in a row.
- Divide that cost by *M*.

For the *add* function that adds to the end of the list, we know the worst case is going to be O(N) because you may have to resize the array and copy over all the elements. However, if the following two things are true about your implementation, then your amortized cost should be O(1):

- Your worst case only happens when the resizing is necessary. *Every other add operation should be O(1). This means NO SHIFTING!*
- You resize by doubling.

We will see the mathematical justification for this when we discuss Stacks and Queues.

Implementation Guidelines

- For some of these methods, you will need to shift items over to make room. However, that is not necessary for all the operations. Instead of shifting, implement the ArrList using a circular array. That means that you can always add or remove from either end in O(1) time (unless it has to be resized.)
- Because of this implementation, the *index* associated with an item in the ArrList will not be guaranteed to be the same as its index in the array. But the index of the array itself should be completely abstracted away from the user.
- I recommend implementing the following private helper methods:
    - `int increment(int i): return (i+1) % array.length. This will help when you need to increment an index in the array because it will wrap around to 0 when it reaches the end of the array.`
    - `ind decrement(int i): does the same thing as increment except it wraps around from 0 to N-1`
    - `void resize(int newCap): resize the array to newCap and copy the items over`
    - `shiftLeft/shiftRight: shifts items either to the left or right; implementation details will vary, but it will help if you write separate methods for these`

Points of Difficulty

- Implementing the wrap-around, circular array can be challenging, but it should be fairly straightforward if you follow the suggestions above.
- Implementing `removeRange` so that its worst case is O(N) will likely be one of the trickier parts to this project. You may be tempted to just do a loop and call `remove` for each of the elements in the range, but this is not a good idea because `remove` is probably going to be shifting elements over each time, and you don't want to be shifting the same elements over and over again. That would make the runtime $O(N^2)$.

    - Note that in the examples below, I am simplifying things assuming the indexes match the array indexes.
    - Example of a bad approach:
        - [0, 1, 2, 3, 4, 5, 6, 7] → removeRange(3, 6)
            - remove(3): [0, 1, 2, 4, 5, 6, 7, _]
            - remove(3): [0, 1, 2, 5, 6, 7, _, _]
            - remove(3): [0, 1, 2, 6, 7, _, _, _]
        - This is bad because it shifted [4, 5, 6, 7] over, then it shifted [5, 6, 7] over, then it shifted [6, 7] over…
        - …and if you did that on a range of size M, that would be M + M-1 + M-2 + … + 1 = M^2.
    - Example of a better approach:
        - [0, 1, 2, 3, 4, 5, 6, 7] → removeRange(3, 6)
        - move 6 to index 3: [0, 1, 2, 6, 4, 5, _, 7]
        - move 7 to index 4: [0, 1, 2, 6, 7, 5, _, _]
        - remove 5: [0, 1, 2, 6, 7, _, _, _]
- Also be careful with the `clear` method, as there are ways to do it efficiently and ways to do it very inefficiently.

Testing

`ArrListTest.java` is provided to help you test your code. Make sure your code works with this file.

Submission Procedure.

To submit, upload the following files to **lectura** and submit them using **turnin**. Note: Lectura is a computer. You need to first transfer your files to that computer before using the turnin command. I have provided two PDFs with some instructions on how you might do this, but it will depend on your system. Once you log in to lectura, you can submit using the following command:

turnin cs345p2 ArrList.java

Upon successful submission, you will see this message:

Turning in:

ArrList.java – OK

All done.

Note: Your submission must be able to run on **lectura**, so make sure you give yourself enough time before the deadline to check that it compiles and runs on lectura and do any necessary debugging. I recommend finishing the project locally 24 hours before the deadline to make sure you have enough time to deal with any submission issues. **I also *strongly recommend* that you make sure you know how to log in to lectura and even practice submitting long before the deadline so that if you have any issues, you can see a TA during office hours for help.**

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.

Grading.

The auto-graded tests are worth 60 points. However there may be additional deductions depending on your solution, such as

- Not following instructions (up to 100% deduction on that part of the assignment)
- Bad Coding Style (up to 10-point deduction total)
- Not satisfying the runtime requirements (up to 50% deduction for that part of the assignment)
- Implementing something in an unnecessarily inefficient way even if it technically meets the runtime requirements (up to 25% deduction for that part of the assignment)
- Improper submission (up to 100% deduction on that part of the assignment)
- Does not compile (up to 100% deduction on that part of the assignment)

Regrade Requests

- We do not enjoy giving zeroes, especially when it is because of a small error, so we do allow regrade requests especially in cases where you received a 0 because of a small compiling issue or something else that can be fixed easily. There is still a standard deduction for most of these issues that is generally equivalent to turning the project in three days late.

- All regrade requests should be submitted according to the guidelines in the syllabus and within the allotted time frame.