Module 1 Assignment: ArrayList

Marcar esta página

Coding Assignment fecha límite Aug 16, 2023 05:50 -05

ArrayLists

Congrats! You have reached the end of the first module in this course! It is now time for you to complete your first coding assignment. For this assignment, you will be implementing an *ArrayList*. Recall that an *ArrayList* is a list data structure backed by an array where all of the data is contiguous and aligned with index 0 of the array. Your *ArrayList* should follow the requirements stated in the javadocs of each method you are to implement.

IMPORTANT:

- **You will be given 5 attempts on this assignment, with a 30 minute cooldown between submissions.**
- **Please run your code before each submission to ensure that there are no formatting errors! If there are formatting errors in your code, your code will not be graded and a submission attempt will be logged. For more information, please review the Vocareum overview below.**

Capacity

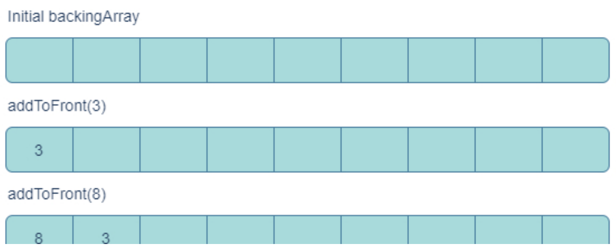
The starting capacity of the *ArrayList* should be the constant `INITIAL_CAPACITY` defined in the file. Reference this constant as-is. Do **not** simply copy the value of this constant. Do **not** change the constant. If, while adding an element, the *ArrayList* does not have enough space, you should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

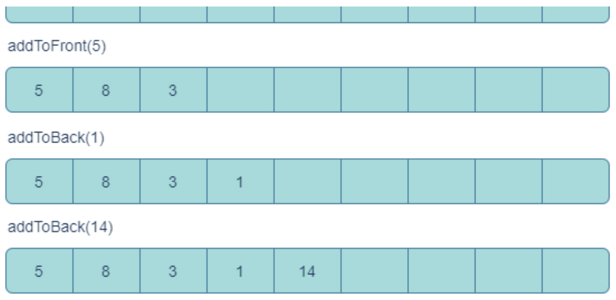
This means that if the initial capacity of the backing array is 9, then after one resize the capacity will be 18, and after two resizes the capacity will be 36. Do **not** resize the backing array when removing elements.

Adding

You will implement two *add()* methods. One will add to the front and one will add to the back. When adding to the front of the list, subsequent elements must be shifted back one position to make room for the new data.

Add() Examples

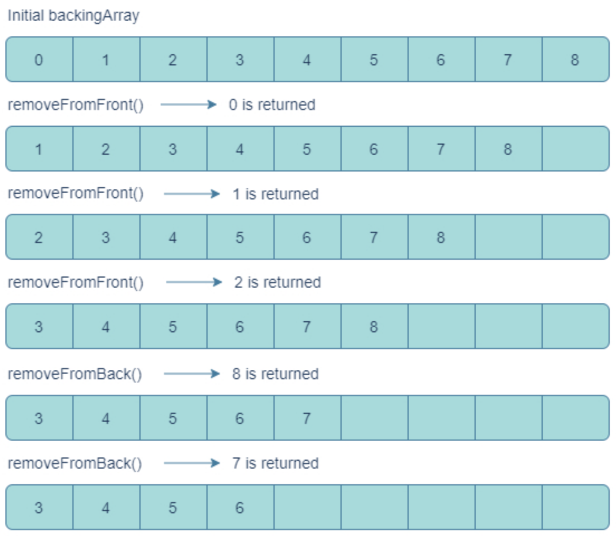




Removing

You will also implement two *remove()* methods. One will remove from the front and one will remove from the back. When removing from the front, 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.

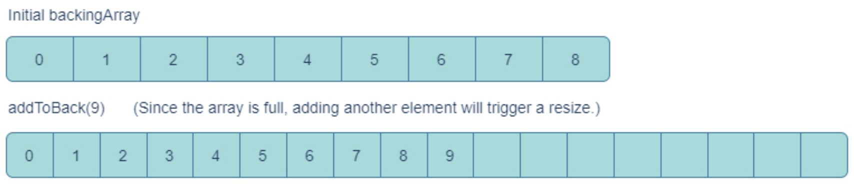
Remove() Examples


Initial backingArray
0 1 2 3 4 5 6 7 8
removeFromFront() → 0 is returned
1 2 3 4 5 6 7 8
removeFromFront() → 1 is returned
2 3 4 5 6 7 8
removeFromFront() → 2 is returned
3 4 5 6 7 8
removeFromBack() → 8 is returned
3 4 5 6 7
removeFromBack() → 7 is returned
3 4 5 6

Resizing

In order to resize your backing array, you will need to create new array of size (2 * old length) and copy all the elements from the old backing array into the new backing array. Don't forget to reassign the *backingArray* variable to the new backing array!

Resize Example


Initial backingArray
0 1 2 3 4 5 6 7 8
addToBack(9) (Since the array is full, adding another element will trigger a resize.)
0 1 2 3 4 5 6 7 8 9

Amortized Efficiency

The efficiency of methods and algorithms in this course is often analyzed using a "per operation" analysis. Please refer to Module 1 - Big O Concepts, for more details. For example, in this assignment, 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 $O(n)$. Also note that since the capacity doubles with each resize, the need of another resize operation decreases as more and more elements are added.

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**.

General Tips

- In *add()* methods, if the passed in data is null, an exception should be thrown (see details in javadocs) and your *ArrayList* should **not** be modified in any way. Where is the most optimal location in the method to check for this situation? Should this check be performed at the beginning or at the end?

- The *size* variable is used to keep track of the number of elements in the *ArrayList*. Whenever data is successfully added/removed from the backing array, the *size* variable should be incremented/decremented appropriately. How can this variable be leveraged to check if your backing array needs to be resized?

- We highly recommend copying the starter code and working in your preferred IDE in order to have access to features such as code completion, auto-formatting, and much more!

Here are general assignment guidelines that should be followed.

- Do not include any package declarations in your classes.
- 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. Instead, exceptions should be thrown as follows: `throw new InsertExceptionHere("Error: some exception was thrown");`
- All helper methods you choose to write should be made private. Recall the idea of Encapsulation in Object-Oriented Programming!
- Do not use anything that would trivialize the assignment. (e.g. Don't import/use java.util.ArrayList for an ArrayList assignment.)
- 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).
- If applicable, 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()`.

Use of the following statements should be avoided at all times.

package	System.arraycopy()	clone()
assert()	Arrays class	Array class
Thread class	Collections class	Collection.toArray()
Reflection APIs	Inner or nested classes	Lambda Expressions

The Vocareum (code editor) interface has six main components:

- The **Drop-Down** in the top left. This lets you choose from multiple available files. Note that this drop-down will only be visible in assignments that require multiple files.
- The **Run** button. This will compile your code and run a file scan. Running your code will not count towards your total allowed submission attempts, therefore you are free to run as many times as needed.
- The **Submit** button. This will compile your code, run a file scan, grade your assignment, and output results to console. Note that for most assignments in this class, you will only be allowed a limited number of submissions. A submission is counted when the submit button is clicked, regardless of whether or not your code can compile or if there are any file issues. Therefore, we **highly recommend** that you run your code before submitting to ensure that there are no issues that will prevent your code from being graded and that every submission attempt will generate meaningful results.
- The **Reset** button. This will revert all your changes and reset your code to the default code template.
- The **Code Window**. This is where you will write your code. For large coding assignments, we highly recommend copying the starter code and working in your preferred IDE to have access to features such as code completion, auto-formatting, and much more!
- The **Output Window**. This window will appear whenever you run or submit your code and will display the output for you to view.

For additional help, please visit the Vocareum information page located in the course information module!