CSCI241 Fall 2020: Lab 4
Due Sunday, October 25th at 9:59pm

1 Overview

In this lab, you'll implement `AList`, an ArrayList-like dynamic array class that automatically expands its underlying storage array as needed. Because this is a useful data structure for storing all manner of data types, you'll be making the AList *generic*, so that it can contain any single type of object.

2 Git and submission for Lab 4

The Github Classroom link for Lab 4 is available in the Lab 4 assignment on Canvas. Clone your repository as usual; be sure to clone it somewhere outside your other lab and assignment repositories to avoid having nested local working copies of different repos. As usual, you will submit your code for this lab by committing and pushing the completed files to your remote Lab 4 repository on GitHub before the deadline.

It is recommended that you `git add` *and* `git commit` *regularly while developing your code; e.g., one or more times per method you implement. When you have something working,* `git push` *your changes to GitHub.*

3 AList Overview

The AList class is modeled after Java's ArrayList class. The functionality you will implement is fairly simple: the AList uses an array to store its contents, keeps track of its own size, and has various accessor and mutator methods. **Notice** that the `size` field is the number of elements in the list, which is not necessarily the same as the `length` field of the underlying storage array (which we'll refer to as the *capacity*).

The type of the underlying storage array is a generic type placeholder, `T`. You'll find that you never need to worry about what type `T` actually refers to, because the AList behavior doesn't depend on the type of the objects being stored. The only time this comes into play is methods that set or return values; instead of a fixed type (e.g., `int`), we simply use the `T` placeholder to refer to whatever type is being stored.

4 Your Tasks

You have four tasks. When you finish implementing each TODO, run the tests using `gradle test` and make sure you're passing the corresponding tests; if not, debug the issue until the test passes.

1. One constructor is provided, which creates an AList with a default underlying storage array size of 8. Implement the one-argument constructor that allows the user to specify the initial capacity of the storage array. Run the tests and make sure you pass the `test00Constructors` test, i.e., you're failing only 4 of the 5 tests.

2. If the size of the AList grows beyond the underlying array's capacity, we need to allocate a new, bigger array. Implement `growIfNeeded` and `resize` according to their specifications. **Use the provided `createArray` method to perform the allocation** in `growIfNeeded`; then, `resize` can change the size of the list and call `growIfNeeded` to expand the capacity as necessary. Your code should now also pass `test10Resize`.

3. Implement `put` and `get` to provide array-like indexing into the AList. Your code should now pass `test20PutGet` and `test21PutGet`.

4. Implement `append` and `pop` to provide easy methods for adding to and removing from the end of the list. Keep in mind that adding to the end of the list may cause `size` to exceed `capacity`. Your code should now pass all five tests.

Rubric

Make sure you have committed your completed AList.java to git and pushed to Github.

Each of the passed JUnit test cases is worth 2 points. Deductions may be made for:

- Problems with submission mechanics
- Code that does not compile or generates runtime errors.
- Poor coding style (e.g. inconsistent indentation, insufficient comments)
- Other assorted failures to follow instructions