Project 1B: ArrayDeque61B

Due: Monday, February 12th, 11:59 PM PT

FAQ

Each assignment will have an FAQ linked at the top. You can also access it by adding "/faq" to the end of the URL. The FAQ for Project 1B is located here.

Note that this project has limited submission tokens. Please see Submit to the Autograder for more details.

Introduction

In Project 1A, we built `LinkedListDeque61B`. Now we'll see a different implementation of the `Deque61B` interface that uses a *backing array*, rather than linked nodes.

By the end of Project 1B, you will...

- Gain an understanding of the implementation of a backing array in data structures.
- Have more experience using testing and test-driven development to verify the correctness of these data structures.

INFO
Check out the Project 1B slides for some additional visually oriented tips.

INFO
Check out the Getting Started Video for overview of spec.

We will provide relatively little scaffolding. In other words, we'll say what you should do, but not how.

INFO

This section assumes you have watched and fully digested the lectures up till the `ArrayList` lecture, Lecture 7.

TASK

For this project, you must work alone! Please carefully read the Policy on Collaboration and Cheating to see what this means exactly. In particular, do not look for solutions online.

DANGER

It should (still) go without saying that you may not use any of the built-in `java.util` data structures in your implementation! The whole point is to build your own versions! There are a few places where you may use specific data structures outside of tests, and we will clearly say where.

Style

As in Project 1A, **we will be enforcing style**. You must follow the style guide, or you will be penalized on the autograder.

You can and should check your style locally with the CS 61B plugin. **We will not remove the velocity limit for failing to check style.**

Getting the Skeleton Files

Follow the instructions in the Assignment Workflow guide to get the skeleton code and open it in IntelliJ. For this project, we will be working in the `proj1b` directory.

You see a `proj1b` directory appear in your repo with the following structure:

```
 proj1b
├── src
│    └── Deque61B.java
└── tests
     └── ArrayDeque61BTest.java
```

If you get some sort of error, STOP and either figure it out by carefully reading the git WTFs or seek help at OH or Ed. You'll potentially save yourself a lot of trouble vs. guess-and-check with git commands. If you find yourself trying to use commands recommended by Google like `force push`, don't. **Don't use force push, even if a post you found on Stack Overflow says to do it!**

You can also watch Professor Hug's demo about how to get started and this video if you encounter some git issues.

Deque: ADT and API

If you need a refresher on `Deque61B`s, refer to the Project 1A spec and the `Deque61B.java` file.

Creating the File

Start by creating a file called `ArrayDeque61B`. This file should be created in the `proj1b/src` directory. To do this, right-click on the `src` directory, navigate to "New -> Java Class", and give it the name `ArrayDeque61B`.

Just like you did in Project 1A We want our `ArrayDeque61B` to be able to hold several different types. To enable this, you should edit the declaration of your class so that it reads:

```
public class ArrayDeque61B<T>
```

Copy

Recall from lecture that it doesn't actually matter if we use `T` or some other string like `ArrayDeque61B<Glerp>`. However, we recommend using `<T>` for consistency with other Java code.

We also want to tell Java that every `ArrayDeque61B` is a `Deque61B`, so that users can write code like `Deque61B<String> lld1 = new ArrayDeque61B<>();`. To enable this, change the declaration of your class so that it reads:

```
public class ArrayDeque61B<T> implements Deque61B<T>
```

Copy

Once you've done this step, you'll likely see a squiggly red line under the entire class declaration. This is because you said that your class implements an interface, but you haven't actually implemented any of the interface methods yet.

Hover over the red line with your mouse, and when the IntelliJ pop-up appears, click the "Implement methods" button. Ensure that all the methods in the list are highlighted, and click "OK". Now, your class should be filled with a bunch of empty method declarations. These are the methods that you'll need to implement for this project!

Lastly, you should create an empty constructor. To do this, add the following code to your file, leaving the constructor blank for now.

```java
public ArrayDeque61B() {
}
```

Copy

Note: You can also generate the constructor by clicking “Code”, then “Generate” then “Constructor”, though I prefer the typing the code yourself approach.

Now you’re ready to get started!

`ArrayDeque61B`

As your second deque implementation, you’ll build the `ArrayDeque61B` class. This deque **must** use a Java array as the backing data structure.

You may add any private helper classes or methods in `ArrayDeque61B.java` if you deem it necessary.

Constructor

You will need to somehow keep track of what array indices hold the deque’s front and back elements. We **strongly recommend** that you treat your array as circular for this exercise. In other words, if your front item is at position `0`, and you `addFirst`, the new front should loop back around to the end of the array (so the new front item in the deque will be the last item in the underlying array). This will result in far fewer headaches than non-circular approaches.

INFO

See the Project 1B demo slides for more details. In particular, note that while the conceptual deque and the array contain the same elements, they do not contain them in the same order.

We recommend using the `floorMod(int a, int b)` method from Java’s built-in `Math` class to assist you in designing a circular approach. Whereas `a % b` might return negative numbers when a is negative, `floorMod(int a, int b)` always return non-negative numbers. In practice, this means that the output will have the same sign as the divisor. Here are a few examples using the `floorMod(int a, int b)` method:

```java
    int value1 = Math.floorMod(16, 16); // value1 == 0
    int value2 = Math.floorMod(-1, 16); // value2 == 15
    int value3 = Math.floorMod(20, 16); // value3 == 4
```

Copy

You can use the `floorMod(int a, int b)` method by adding the following import statement to the top of your file: `import java.lang.Math;`.

WARNING

You cannot create an array of generics (e.g. `new T[1000]`) in Java for reasons beyond the scope of this course. You will instead need to use the syntax `(T[]) new Object[1000]`.

TASK

Declare the necessary instance variables, and implement the constructor.

The starting size of your backing array **must** be `8`.

`addFirst` and `addLast`

As before, implement `addFirst` and `addLast`. These two methods **must not** use looping or recursion. A single add operation must take “constant time,” that is, adding an element should take approximately the same amount of time no matter how large the deque is (with one exception). This means that you cannot use loops that iterate through all / most elements of the deque.

Resizing Up

WARNING

We recommend you complete the other methods first, verify that they are working correctly without resizing, and come back to resizing later.

The exception to the “constant time” requirement is when the array fills, and you need to “resize” to have enough space to add the element. In this case, you can take “linear time” to resize the array before adding the element.

Correctly resizing your array is very tricky, and will require some deep thought. Try drawing out various approaches by hand. It may take you quite some time to come up with the right approach, and we encourage you to debate the big ideas with your fellow students or TAs. Make sure that your actual implementation is **by you alone**.

Make sure to resize by a geometric factor.

DANGER

We **do not** recommend using `arraycopy` with a circular implementation. It will work, but results in a significantly more complex (and harder to debug!) implementation than necessary.

Instead, we suggest thinking forward to how you might implement `get` and using a `for` loop in some way.

TASK

Implement `addFirst` and `addLast`, and write tests to verify that they are correct. Make sure to add enough elements so that your backing array resizes! For more info on resizing, check out these slides.

`get`

Unlike in `LinkedListDeque61B`, this method must take **constant time**.

As before, `get` should return `null` when the index is invalid (too large or negative). You should disregard the skeleton code comments for `Deque61B.java` for this case.

TASK

After you've written tests and verified that they fail, implement `get`.

`isEmpty` and `size`

These two methods must take **constant time**. That is, the time it takes to for either method to finish execution should not depend on how many elements are in the deque.

TASK

Write tests for the `isEmpty` and `size` methods, and check that they fail. Then, implement the methods.

`toList`

`toList` will continue to be useful to test your `Deque61B`.

Write the `toList` method. The first line of the method should be something like `List<T> returnList = new ArrayList<>()`. **This is one location where you are allowed to use a Java data structure.**

WARNING

Some later methods might seem easy if you use `toList`. **You may not call `toList` inside** `ArrayDeque61B`; there is a test that checks for this.

INFO

Hint One of the other methods may be helpful for implementing this method.

TASK

Implement `toList`. You are not given tests this time, so you will need to write them!

All that's left is to test and implement all the remaining methods. For the rest of this project, we'll describe our suggested steps at a high level. We **strongly encourage** you to follow the remaining steps in the order given. In particular, **write tests before you implement the method's functionality.** This is called "test-driven development," and helps ensure that you know what your methods are supposed to do before you do them.

`removeFirst` and `removeLast`

Lastly, write some tests that test the behavior of `removeFirst` and `removeLast`, and again ensure that the tests fail.

Do not maintain references to items that are no longer in the deque.

`removeFirst` and `removeLast` **may not** use looping or recursion. Like `addFirst` and `addLast`, these operations must take "constant time." Refer to the section on writing `addFirst` and `addLast` for more information on what this means.

Resizing Down

The amount of memory that your program uses at any given time must be proportional to the number of items. For example, if you add 10,000 items to the deque, and then remove 9,999 items, you shouldn't still be using an array that can hold 10,000 items. For arrays of length 16 or more, your usage factor should always be at least 25%. This means that before performing a remove operation, if the number of elements in the array is at or under 25% the length of the array, you should resize the array down. For arrays length 15 or less, your usage factor can be arbitrarily low.

DANGER

We, again, **do not** recommend using `arraycopy` with a circular implementation. If you followed our advice above to use a `for` loop to resize up, resizing down should look **very similar** to resizing up (perhaps a helper method?).

TASK

After you've written tests and verified that they fail, implement `removeFirst` and `removeLast`.

DANGER

For the intended experience, follow these steps in order. If you do something else and ask us for help, we will refer you back to these steps.

`getRecursive`

Although we are not using a linked list anymore for this project, it is still required to implement this method to keep consistent with our interface. This method technically shouldn't be in the interface, but it's here to make testing nice. You can just use this code block for it:

```
@Override
public T getRecursive(int index) {
    throw new UnsupportedOperationException("No need to implement getRecursive for proj 1
}
```

Copy

TASK

"Implement" `getRecursive`.

Writing Tests

Refer to the Project 1A spec for a review of how to write tests. Similar to Project 1A, you will be scored on the coverage of your unit tests for Project 1B. You might find some of your tests from Project 1A to be reusable in this project; don't be afraid to copy them over!

Suggestions

- Try to get everything working for a fixed-size array first. This would be good point to start to familiarize yourself.
- Once you are confident working solution for a fixed-size array, try resizing - consider having a helper method for it!
- **DO NOT** modify `Deque61B` interface

Submit to the Autograder

Once you've written local tests and passed them, try submitting to the autograder. You may or may not pass everything.

- If you fail any of the coverage tests, it means that there is a case that your local tests did not cover. The autograder test name and the test coverage component will give you hints towards the missing case.

- If you fail a correctness test, this means that there is a case that your local tests did not cover, despite having sufficient coverage for flags. This is **expected**. Coverage flags are an approximation! They also do not provide describe every single behavior that needs to be tested, nor do they guarantee that you assert everything. Here is a list of them!
- If you fail any of the timing tests, it means that your implementation does not meet the timing constraints described above.
- You will have a token limit of 4 tokens every 24 hours. **We will not reinstate tokens for failing to add/commit/push your code, run style, etc.**
- You may find messages in the autograder response that look something like this: `WARNING: A terminally deprecated method in java.lang.System has been called`. You can safely ignore any line tagged as a `WARNING`.

Scoring

This project, similar to Project 0, is divided into individual components, each of which you must implement *completely correctly* to receive credit.

1. **Adding (25%)**: Correctly implement `addFirst`, `addLast`, and `toList`.
2. `isEmpty`**,** `size` **(5%)**: Correctly implement `isEmpty` and `size` with add methods working.
3. `get` **(10%)**: Correctly implement `get`.
4. **Removing (30%)**: Correctly implement `removeFirst` and `removeLast`.
5. **Memory (20%)**: Correctly implement resizing so that you do not use too much memory.

Additionally, there is a **test coverage (10%)** component. We will run your tests against a staff solution, and check how many scenarios and edge cases are tested. You can receive partial credit for this component. Here is a list of them!