



Project 1C: Deque Enhancements

Due: February 22nd

Introduction

Style

Getting the Skeleton Files

Object Methods

`iterator()`

`equals()`

`toString()`

Testing The Object Methods

MaxArrayDeque

Guitar Hero

The GH2 Package

GuitarString

Why It Works

GuitarHeroLite

The Birds

Other Possibilities for Further Enrichment

Submission

Scoring

Credits

Due: February 22nd

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 1C is located [here](#).

Introduction

In Project 1A, we built `LinkedListDeque` and in Project 1B, we built `ArrayDeque`. Now we'll see a different implementation: `MaxArrayDeque`! This part of the project will provide some enhancements to your previous `ArrayDeque` and `LinkedListDeque`, and also bring everything together into an application of your newly-built data structure.

By the end of Project 1C, you will complete the following:

- Write the `iterator()`, `equals()`, and `toString()` methods for `LinkedListDeque.java` and `ArrayDeque.java`.
- Implement `MaxArrayDeque.java`.
- Finish the `GuitarHero` tasks.

! This section assumes you have watched and fully digested the lectures up till the Iterators, Object Methods lecture, Lecture 12.

Style

As in Project 1B, **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 `proj1c` directory.

You see a `proj1c` directory appear in your repo with the following structure:

```
proj1c
├── src
│   ├── deque
│   │   ├── ArrayDeque.java
│   │   ├── Deque.java
│   │   └── LinkedListDeque.java
│   └── gh2
│       ├── GuitarHeroLite.java
│       ├── GuitarPlayer.java
│       ├── GuitarString.java
│       └── TTFAF.java
└── tests
    ├── MaxArrayDequeTest.java
    └── TestGuitarString.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.

Object Methods

If you'd like, you can follow the steps in this short [video guide](#) to help you get set up for Project 1C!

In order to implement the following methods, you should start by copying and pasting your Project 1A and Project 1B implementations of `LinkedListDeque` and `ArrayDeque` into the relevant files in your `proj1c` directory.

❗ Important: Because of the way that the `Deque` interfaces were structured in Projects 1A and 1B, you'll need to implement the `getRecursive()` method in `ArrayDeque` after copy-pasting it. **If you don't implement this method, both the autograder and your own code will not compile.** This doesn't need to be an actual implementation of the method, since we won't test it. Instead, it can just look like the code snippet below (feel free to copy-paste this snippet directly into your file).

```
@Override
public T getRecursive(int index) {
    return get(index);
}
```

iterator()

One shortcoming of our `Deque` interface is that it can not be iterated over. That is, the code below fails to compile with the error “foreach not applicable to type”.

```
Deque<String> lld1 = new LinkedListDeque<>();

lld1.addLast("front");
lld1.addLast("middle");
lld1.addLast("back");
for (String s : lld1) {
    System.out.println(s);
}
```

Similarly, if we try to write a test that our `Deque` contains a specific set of items, we'll also get a compile error, in this case: “Cannot resolve method containsExactly in Subject”.

```

public void addLastTestBasicWithoutToList() {
    Deque<String> lld1 = new LinkedListDeque<>();

    lld1.addLast("front"); // after this call we expect: ["front"]
    lld1.addLast("middle"); // after this call we expect: ["front",
    lld1.addLast("back"); // after this call we expect: ["front", "
    assertThat(lld1).containsExactly("front", "middle", "back");
}

```



Again the issue is that our item cannot be iterated over. The `Truth` library works by iterating over our object (as in the first example), but our `LinkedListDeque` does not support iteration.


To fix this, you should first modify the `Deque` interface so that the declaration reads:

```

public interface Deque<T> extends Iterable<T> {

```

Next, implement the `iterator()` method using the techniques described in lecture 12.

 **Task:** Implement the `iterator()` method in both `LinkedListDeque` and `ArrayDeque` according to lecture.

equals()

Consider the following code:

```

@Test
public void testEqualDeque() {
    Deque<String> lld1 = new LinkedListDeque<>();
    Deque<String> lld2 = new LinkedListDeque<>();

    lld1.addLast("front");
    lld1.addLast("middle");
    lld1.addLast("back");

```

```

l1d2.addLast("front");
l1d2.addLast("middle");
l1d2.addLast("back");

assertThat(l1d1).isEqualTo(l1d2);
}

```

If we run this code, we see that we fail the test, with the following message:

```

expected: [front, middle, back]
but was : (non-equal instance of same class with same string repres

```



The issue is that the `Truth` library is using the `equals` method of the `LinkedListDeque` class. The default implementation is given by the [code below](#):


```


public boolean equals(Object obj) {
    return (this == obj);
}

```

That is, the `equals` method simply checks to see if the addresses of the two objects are the same.

Override the `equals` method in the `ArrayDeque` and `LinkedListDeque` classes. For guidance on writing an `equals` method, see the [lecture slides](#) or the [lecture code repository](#).

 **Task:** Override the `equals()` method in the `LinkedListDeque` and `ArrayDeque` classes.

 **Important:** You should not use `getClass`, and there's no need to do any casting in your `equals` method. That is, you shouldn't be doing `(ArrayDeque) o`. Such `equals` methods are old fashioned and overly complex.

❗ Important: Make sure you use the `@Override` tag when overriding methods. A common mistake in student code is to try to override `equals(ArrayList<T> other)` rather than `equals(Object other)`. Using the optional `@Override` tag will prevent your code from compiling if you make this mistake. `@Override` is a great safety net.

toString()

Consider the code below, which prints out a `LinkedListDeque`.

```
Deque<String> lld1 = new LinkedListDeque<>();

lld1.addLast("front");
lld1.addLast("middle");
lld1.addLast("back");

System.out.println(lld1);
```


This code outputs something like

`deque.proj1a.LinkedListDeque@1a04f701`. This is because the print statement implicitly calls the `LinkedListDeque` `toString` method. Since you didn't override this method, it uses the default, which is given by the code below (you don't need to understand how this code works).

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```



In turn the `hashCode` method, which you have also not overridden, simply returns the address of the object, which in the example above was `1a04f701`.

 **Task:** Override the `toString()` method in the `LinkedListDeque` and `ArrayDeque` classes, such that the code above prints out `[front, middle, back]`.

! Hint: Java's implementation of the `List` interface has a `toString` method.

! Hint: There is a one line solution (see hint 1).

! Hint: Your implementation for `LinkedListDeque` and `ArrayDeque` should be exactly the same.

Note: You might ask why we're implementing the same method in two classes rather than providing a `default` method in the `Deque` interface. Interfaces are not allowed to provide `default` methods that override `Object` methods. For more see <https://stackoverflow.com/questions/24595266/why-is-it-not-allowed-add-tostring-to-interface-as-default-method>.

Testing The Object Methods

We haven't provided you with test files for these three object methods; however, we strongly encourage you to use the techniques you learned from projects 1A and 1B to write your own tests. You can structure these tests however you'd like, since we won't be testing them. One possible (and suggested) structure is to create two new files in the `tests` directory called `LinkedListDequeTest` and `ArrayDequeTest`, similar to the ones we gave you in 1A and 1B.

MaxArrayDeque

After you've fully implemented your `ArrayDeque` and tested its correctness, you will now build the `MaxArrayDeque`. A `MaxArrayDeque` has all the methods that an `ArrayDeque` has, but it also has 2 additional methods and a new constructor:

- `public MaxArrayDeque(Comparator<T> c)`: creates a `MaxArrayDeque` with the given `Comparator`.

- `public T max()`: returns the maximum element in the deque as governed by the previously given `Comparator`. If the `MaxArrayDeque` is empty, simply return `null`.
- `public T max(Comparator<T> c)`: returns the maximum element in the deque as governed by the parameter `Comparator c`. If the `MaxArrayDeque` is empty, simply return `null`.

The `MaxArrayDeque` can either tell you the max element in itself by using the `Comparator<T>` given to it in the constructor, or an arbitrary `Comparator<T>` that is different from the one given in the constructor.

We do not care about the `equals(Object o)` method of this class, so feel free to define it however you think is most appropriate. We will not test this method.

❗ If you find yourself starting off by copying your entire `ArrayDeque` implementation in a `MaxArrayDeque` file, then you're doing it wrong. This is an exercise in clean code, and redundancy is one of our worst enemies when battling complexity! For a hint, re-read the second sentence of this section above.

 **Task:** Fill out the `MaxArrayDeque.java` file according to the API above.

There are no runtime requirements on these additional methods, we only care about the correctness of your answer. Sometimes, there might be multiple elements in the `MaxArrayDeque` that are all equal and hence all the max: in this case, you can return any of them and they will be considered correct.

You should write tests for this part as well! You'll likely be creating multiple `Comparator<T>` classes to test your code: this is the point! To get practice using `Comparator` objects to do something useful (find the maximum element) and to get practice writing your own `Comparator` classes. You will not be turning in these tests, but we still highly suggest making them for your sake.

You will not use the `MaxArrayDeque` you made for the next part; it'll be in an isolated exercise.

Guitar Hero

In this part of the project, we will create another package for generating synthesized musical instruments using the `deque` package we just made. We'll get the opportunity to use our data structure for implementing an algorithm that allows us to simulate the plucking of a guitar string.

The GH2 Package

The `gh2` package has just one primary component that you will edit:

- `GuitarString`, a class which uses an `Deque<Double>` to implement the [Karplus-Strong algorithm](#) to synthesize a guitar string sound.

We've provided you with skeleton code for `GuitarString` which is where you will use your `deque` package that you made in the first part of this project.

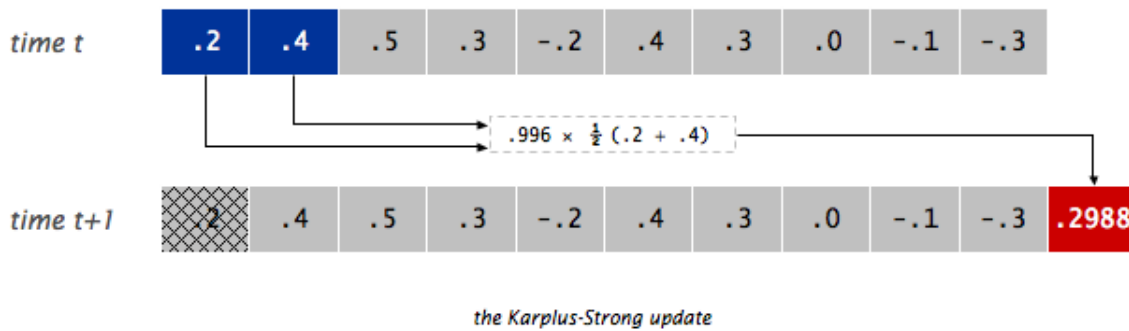
GuitarString

We want to finish the `GuitarString` file, which should use the `deque` package to replicate the sound of a plucked string. We'll be using the Karplus-Strong algorithm, which is quite easy to implement with a `Deque`.

The Karplus-Algorithm is simply the following three steps:

1. Replace every item in a `Deque` with random noise (`double` values between -0.5 and 0.5).
2. Remove the front double in the `Deque` and average it with the next double in the `Deque` (hint: use `removeFirst()` and `get()`) multiplied by an energy decay factor of 0.996 (we'll call this entire quantity `newDouble`). Then, add `newDouble` to the back of the `Deque`.
3. Play the `double` (`newDouble`) that you dequeued in step 2. Go back to step 2 (and repeat forever).

Or visually, if the `Deque` is as shown on the top, we'd remove the 0.2, combine it with the 0.4 to form 0.2988, add the 0.2988, and play the 0.2.



You can play a `double` value with the `StdAudio.play` method. For example `StdAudio.play(0.333)` will tell the diaphragm of your speaker to extend itself to 1/3rd of its total reach, `StdAudio.play(-0.9)` will tell it to stretch its little heart backwards almost as far as it can reach. Movement of the speaker diaphragm displaces air, and if you displace air in nice patterns, these disruptions will be interpreted by your consciousness as pleasing thanks to billions of years of evolution. See [this page](#) for more. If you simply do `StdAudio.play(0.9)` and never play anything again, the diaphragm shown in the image would just be sitting still 9/10ths of the way forwards.

Complete `GuitarString.java` so that it implements steps 1 and 2 of the Karplus-Strong algorithm. Note that you will have to fill your `Deque` buffer with zeros in the `GuitarString` constructor. Step 3 will be done by the client of the `GuitarString` class.

⚠ Do not call `StdAudio.play` in `GuitarString.java`. This will cause the autograder to break. `GuitarPlayer.java` does this for you already.

i Make sure to add the libraries, as usual, otherwise IntelliJ won't be able to find `StdAudio`.

For example, the provided `TestGuitarString` class provides a sample test `testPluckTheAString` that attempts to play an A-note on a guitar string. If you run the test should hear an A-note when you run this test. If you don't, you should try the `testTic` method and debug from there. Consider adding a `print` or `toString` method to `GuitarString.java` that will help you see what's going on between tics.

Note: we've said `Deque` here, but not specified which `Deque` implementation to use. That is because we only need those operations `addLast`, `removeFirst`, and `get` and we know that classes that implement `Deque` have them. So you are free to choose either the `LinkedListDeque` for the actual implementation, or the `ArrayDeque`. For an optional (but highly suggested) exercise, think about the tradeoffs with using one vs the other and discuss with your friends what you think the better choice is, or if they're both equally fine choices.

Why It Works

The two primary components that make the Karplus-Strong algorithm work are the ring buffer feedback mechanism and the averaging operation.

- **The ring buffer feedback mechanism.** The ring buffer models the medium (a string tied down at both ends) in which the energy travels back and forth. The length of the ring buffer determines the fundamental frequency of the resulting sound. Sonically, the feedback mechanism reinforces only the fundamental frequency and its harmonics (frequencies at integer multiples of the fundamental). The energy decay factor (.996 in this case) models the slight dissipation in energy as the wave makes a round trip through the string.
 - **The averaging operation.** The averaging operation serves as a gentle low-pass filter (which removes higher frequencies while allowing lower frequencies to pass, hence the name). Because it is in the path of the feedback, this has the effect of gradually attenuating the higher harmonics while keeping the lower ones, which corresponds closely with how a plucked guitar string sounds.
-

GuitarHeroLite

You should now also be able to use the `GuitarHeroLite` class. Running it will provide a graphical interface, allowing the user (you!) to interactively play sounds using the `gh2` package's `GuitarString` class.

The Birds

To earn “The Birds”, you must create `GuitarHero` and also implement at least one additional instrument.

Consider creating a program `GuitarHero` that is similar to `GuitarHeroLite`, but supports a total of 37 notes on the chromatic scale from 110Hz to 880Hz. Use the following 37 keys to represent the keyboard, from lowest note to highest note:

```
String keyboard="q2we4r5ty7u8i9op-=[zxdcfvgbnjmk,.;/' ";
```

This keyboard arrangement imitates a piano keyboard: The “white keys” are on the qwerty and zxcv rows and the “black keys” on the 12345 and asdf rows of the keyboard.

The i th character of the string keyboard corresponds to a frequency of $440 \cdot 2^{(i-24)/12}$, so that the character ‘q’ is 110Hz, ‘i’ is 220Hz, ‘v’ is 440Hz, and ‘ ’ is 880Hz. Don’t even think of including 37 individual `GuitarString` variables or a 37-way if statement! Instead, create an array of 37 `GuitarString` objects and use `keyboard.indexOf(key)` to figure out which key was typed. Make sure your program does not crash if a key is pressed that does not correspond to one of your 37 notes.

- Harp strings: Create a `Harp` class in the `gh2` package. Flipping the sign of the new value before enqueueing it in `tic()` will change the sound from guitar-like to harp-like. You may want to play with the decay factors to improve the realism, and adjust the buffer sizes by a factor of two since the natural resonance frequency is cut in half by the `tic()` change.
- Drums: Create a `Drum` class in the `gh2` package. Flipping the sign of a new value with probability 0.5 before enqueueing it in `tic()` will produce a drum sound. A decay factor of 1.0 (no decay) will yield a better sound, and you will need to adjust the set of frequencies used.
- Other: Try inventing a new instrument.

Other Possibilities for Further Enrichment

- Guitars play each note on one of 6 physical strings. To simulate this you can divide your `GuitarString` instances into 6 groups, and when

a string is plucked, zero out all other strings in that group.

- Pianos come with a damper pedal which can be used to make the strings stationary. You can implement this by, on iterations where a certain key (such as Shift) is held down, changing the decay factor.
- While we have used equal temperament, the ear finds it more pleasing when musical intervals follow the small fractions in the just intonation system. For example, when a musician uses a brass instrument to play a perfect fifth harmonically, the ratio of frequencies is $3/2 = 1.5$ rather than $27/12 \sim 1.498$. Write a program where each successive pair of notes has just intonation.

To earn “The Birds”, create a short video demo and fill out [this Google Form](#).

Submission

To submit the project, add and commit your files, then push to your remote repository. Then, go to the relevant assignment on Gradescope and submit there.

The autograder for this assignment will have the following velocity limiting scheme:

- From the release of the project to 10:00PM on 2/22/2023, you will have 6 tokens; each of these tokens will refresh every 24 hours.
 - From 10:00PM to 11:59PM on 2/22/2023 (the last 2 hours before the deadline), you will get 4 tokens; each of these tokens will refresh every 15 minutes. -
-

Scoring

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

1. `LinkedListDeque` **Object Methods (20%)**: Correctly implement `iterator`, `equals`, and `toString` in `LinkedListDeque`.
2. `ArrayDeque` **Object Methods (20%)**: Correctly implement `iterator`, `equals`, and `toString` in `ArrayDeque`.

3. `MaxArrayDeque` **Functionality (5%)**: Ensure your `MaxArrayDeque` correctly runs all the methods in the `Deque` interface.
4. `MaxArrayDeque` **Max (35%)**: Correctly implement `max` in `MaxArrayDeque`.
5. `GuitarString` **(20%)**: Correctly implement the `GuitarString` client class.

In total, Project 1c is worth 512 points.

Credits

- Ring buffer figures from [Wikipedia](#).
- This assignment adapted from [Kevin Wayne's Guitar Heroine](#) assignment.

Last built: 2023-10-26 18:40 UTC