Learning Activity: Stack and Queue Use, Backmasking

Back

Objectives

- Use the stack and queue implementations you have created.

The goal of this LA is to use your implementations of the stack and queue interfaces for the data structures.

Background

In this LA, you will work with backmasking of short audio clips. *Backmasking* is the technique of playing an audio track backward. Its use was popularized by the Beatles and used rather extensively at that period; other artists using backmasking include Jimi Hendrix and Ozzie Ozbourne. Historically, this was done by literally reversing the magnetic tape that held that track. For this assignment, we will use digital signal processing to accomplish the backmasking.

To manipulate the sound clips, they will be converted to DAT (digital audio tape) format. DAT is a textual format, which simplifies the manipulation.

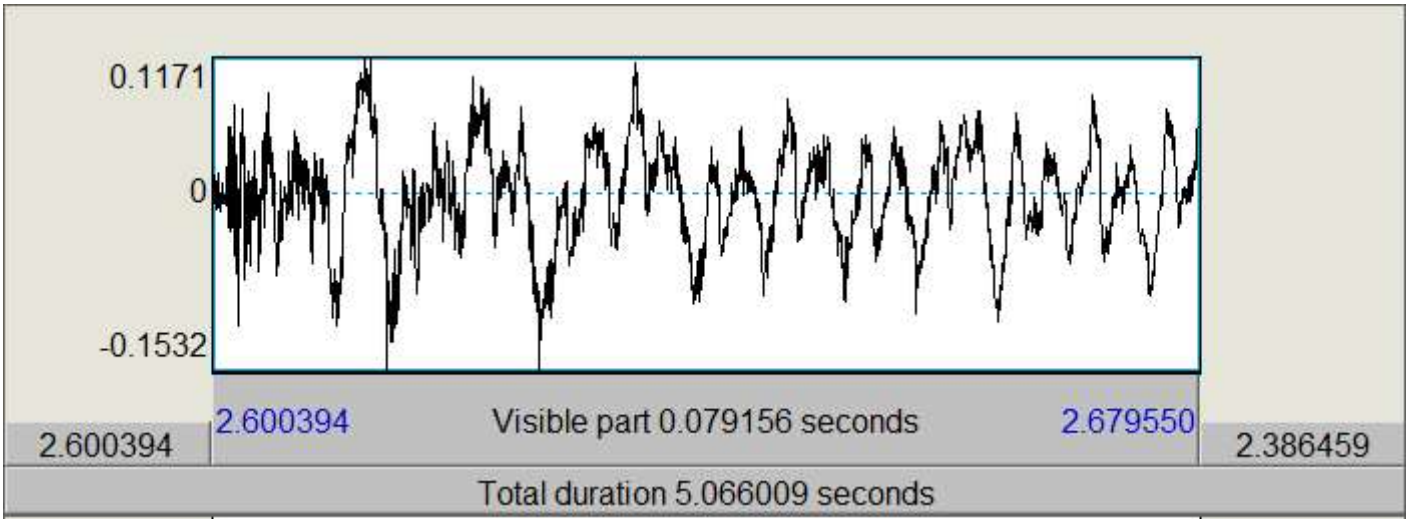
DAT Format

DAT format was developed by Sony in the 80s for capturing the information in a audio signal. As the name implies, the data is stored in digital format, rather than analog. Here are some examples of data taken from a DAT file, bot.dat, which is used in the video clip. Here is the beginning of the file:

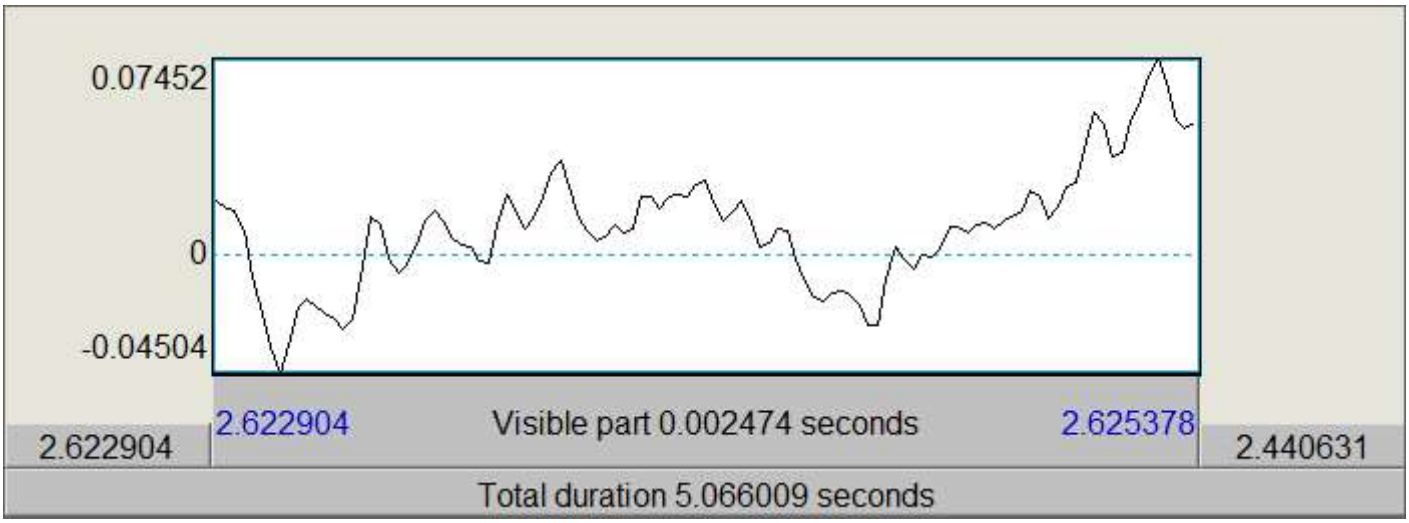
```
; Sample Rate 44100
; Channels 1
             0               0
   2.2675737e-05             0
   4.5351474e-05             0
   6.8027211e-05             0
   9.0702948e-05             0
   0.00011337868             0
   0.00013605442             0
```

The file starts out with some header information, indicated by an initial semicolon. The sampling rate of 44.1 kHz is the typical sampling rate for CDs, since it will capture the top end of human hearing, 22.05 kHz. This recording contains a single audio channel, that is, it's monaural.

Before we start to explain the columns of numbers, let's look at an example of the real sound file. Here is a very short segment of bot.wav displayed in using a digital signal processing utility named Praat. Yes, that jagged line is the sound wave.

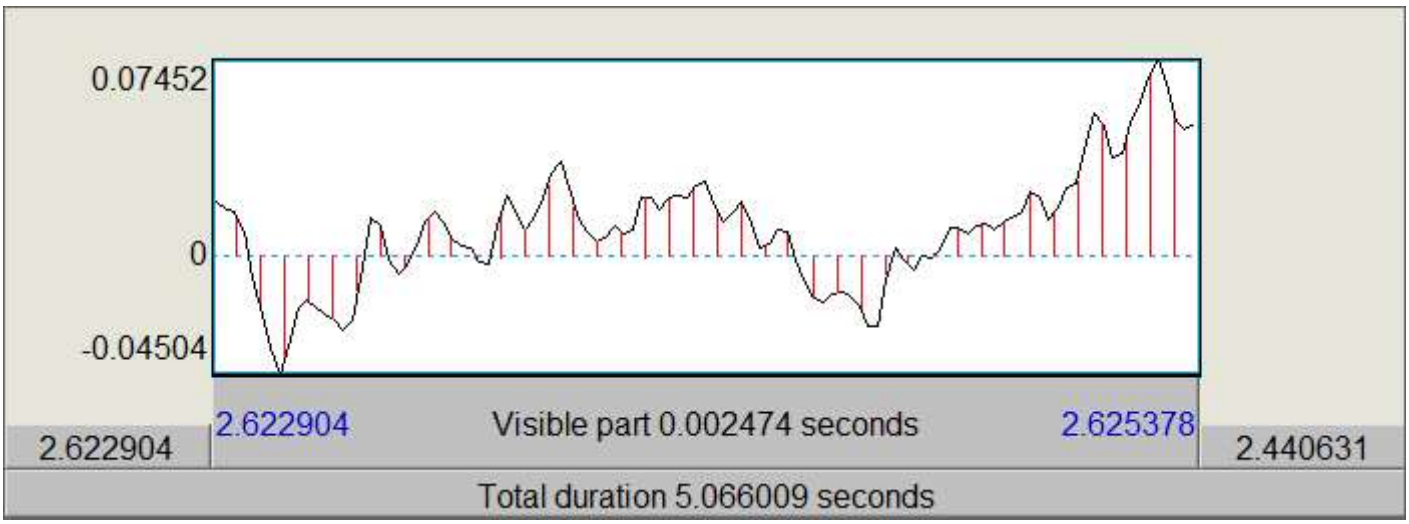


Let's zoom in a lot more.



In this graph, time is the horizontal axis; the vertical axis is a measure of the "displacement" caused by the sound wave. Remember that sound traveling in air is the compression and rarefaction of the air itself.

This is an analog signal. To capture it in a digital format, the "displacement" values are sampled at regular intervals, the sampling rate. Here it is graphically. (Not a real graph of sampling at 44.1 kHz, but an illustration.)



As you can see, the process of sampling loses information. The higher the sampling rate, the more closely we can describe the original wave form. Human hearing caps out at 22.05 kHz though, so a sampling rate of 44.1 kHz will capture sound at that frequency (the Nyquist frequency).

Now, let's look at that sample from the beginning of the dat file again.

```
; Sample Rate 44100
; Channels 1
              0               0
   2.2675737e-05              0
   4.5351474e-05              0
   6.8027211e-05              0
   9.0702948e-05              0
   0.00011337868              0
   0.00013605442              0
```

The first column of numbers give the time signature of the sample; the second column, the "displacement" value. As you can see, the file starts out in silence, no "displacement" from still air.

And here is some data from later in the same file:

```
2.6234921     0.0066223145
2.6235147     0.0039367676
2.6235374     0.0032348633
2.6235601    -0.0022277832
2.6235828    -0.0025634766
2.6236054      0.011291504
2.6236281      0.023223877
2.6236508       0.01751709
```

You can see how data is lost in the process, but the lost information represents frequencies that are beyond the range of human hearing.

The sox Utility

To create a DAT version of a sound clip, you can use sox (SOund eXchange) or some equivalent utility. Download version 14.4.1 of sox:

- sox-14.4.1a-win32.exe (for Windows)
- sox-14.4.1-macosx.zip (for OS-X)

Go to http://sourceforge.net/projects/sox/ to access other versions and formats of this utility. However, the current version, 14.4.2, displayed a problem with the conversion back into the wav format. This is why we're using version 14.4.1 for this exercise.

The sox utility is a command-line application with a number of options and capabilities. We will use only the simplest ones to change format. The command is as follows, where the formats are indicated by the file extension.

```
sox current-format desired-format
```

Let's change bot.wav into DAT format.

In Windows, you can use:

```
sox bot.wav bot.dat
```

In Unix-like systems, such as OS-X, you can use:

```
./sox bot.wav bot.dat
```

This assumes that the sox executable file and bot.wav are in the current working directory.

Needless to say, analogous commands can be used to convert in the other direction, from DAT format back to a sound file.

In Windows:

```
sox bot.dat bot.wav
```

In Unix-like systems, such as OS-X:

```
./sox bot.dat bot.wav
```

Instructions

This project is split into several phrases:

1. Backmasking application (Check Level)
2. Tone generation (Plus Level)

These phases are a trifle out of order, but this is the recommended approach to completing this assignment.

Minus Level - Using java.util.Data Structures

If you do not have working stack and queue implementations from the previous PAs, you may use data structures from the standard library, specifically the java.util package.

Check Level - Using Your Own Stack and Queue

For the Check Level of this LA, you will use your implementations for stack and queue.

Do NOT use any data structures from the standard library for this assignment. In fact, the only resource from java.util you may use is java.util.Scanner. You may use the following import declaration:

```
import java.util.Scanner;
```

Do NOT use the declaration:

```
import java.util.*;
```

This declaration or the use of any resource from java.util other than Scanner will cause the LA to be graded at the Minus Level.

Implementing Backmasking

Using the appropriate implementations of the stack and queue, implement backmasking. Name the application class **csc143.sound.Backmask**.

Install sox (URL given above). Download the data files:

- sample1.wav

- sample2.wav
- sample3.wav
- mystery.wav

Here are the examples files used in the video.

- bot.wav
- botrev.wav
- bot.dat
- botrev.dat

Create an application that will backmask an audio clip in DAT format. Algorithmically, the backmasking process is straightforward. The input and output file formats are textual, so, they can be read using java.util.Scanner, java.io.BufferedReader, or java.io.BufferedInputStream. The output file can be created using java.io.PrintWriter or java.io.PrintStream. The two header lines can be copied directly from the input file to the output file. The data lines need to be manipulated slightly. The time values (first column) are to be printed out in the same order that they appear in the input file, where the displacement values (second column) are to be printed in reverse order. Use your data structures to save these values as you read them in. Store the time values in a queue and the displacement values in a stack. These values can be stored as String. [participation fodder: Why can we use String?] [more fodder: Wouldn't it be better to use Double, with autoboxing and autounboxing?] After you have read all of the values in, you can simply remove from the stack and queue to write out the values in the desired order. To get "pretty printing", you can use one of the formatting methods of PrintStream or PrintWriter.

Use the sox utility and your application to "unbackmask" the backmasked sound files: sample1, sample2, sample3, and mystery. Create an ASCII, plain-text file with the identification of the reversed sound clips. They are all taken from American movies. In the report file, remember to include the standard header information with your name and this LA designation. For the sound clips, identify which clip (sample1, sample2, sample3, mystery) and the text, For a little extra credit, include the name of the movie from which the clip is taken and the speaker (character who spoke the line). Do not discuss the contents of these files on the discussion board.

You can also test your backmasking application by using sample.dat as generated by MakeDat.java, as shown in the video. You won't be able to identify the source of the sound clips, but you will be able to verify that your program is working.

The interface for your application shall be an executable file (`public static void main`) with command-line arguments. It will take two arguments: the name of the input DAT file, the name of the output DAT file. If the wrong number of command-line arguments are given, report the error using System.err and also give usage instructions. (This can be as simple as "usage: java appname input-file output-file", where appname is the name of your application class file.) If either the input file or the output file cannot be opened, report an error using System.err and terminate the application.

Tone Generation - Plus Level

For the extra-credit points of the Plus Level, create a separate application that generates a DAT file which plays a C major scale: C4, D4, E4, F4, G4, A4, B4, C5. Each note should be approximately 0.25 seconds long.

If you want to calculate the values, an octave is the doubling of the frequency. One octave is divided into twelve equal half-steps (in equal temperament). So, a one half-step displacement is multiplying the initial frequency by the twelfth root of two. The displacement of a whole step, that is, two half steps, comes from multiplying the initial frequency by the sixth root of two. In the major scale there are half steps between the third and four tones and between the seventh and eighth tones. In our C-major scale, the half step occur between E4 and F4 and between B4 and C5. All the other intervals in the scale are whole steps. In modern music, A4 has a frequency of 440 Hz.

If you don't want to calculate the frequency values, you can use the following table:

- C4 — 261.63
- D4 — 293.66
- E4 — 329.63
- F4 — 349.23
- G4 — 392.00
- A4 — 440.00
- B4 — 493.88
- C5 — 523.55

The sine wave is a "pure" sound. It's sufficient for this application. Since we are not dealing with any high frequency components to the sound, we can lower the sampling rate with no loss of fidelity. A sampling rate of 8,192 will be more than sufficient.

In the best of all possible worlds, the generated sound wave should not "jump". That is a plot of the wav should not show "discontinuities". This can be accomplished quite easily by insuring that the generated sound wave is composed of complete cycles of the sine wave. (Yes, this is why the write-up states, "Each note should be *approximately* 0.25 seconds long.")

Create a separate Java application for the Plus Level. It shall create the file **scale.dat**.

Deliverables

For the Minus work, two (2) files:

- One (1) Java archive file (.jar) containing source code and bytecode for csc143.sound.Backmask (entry point)
- One (1) ASCII, plain-text report file (.txt) identifying the "unbackmasked" sound clips

For the Check work, two (2) files:

- one (1) Java archive file (.jar) containing source code and bytecode for:
  - csc143.sound.Backmask (entry point)
  - the implementations of the csc143.data_structures classes you used
  - Make sure that the archive file is complete. Missing classes will result in a deduction. However, do not include unneeded files in the archive file. This will also result in a deduction.
- One (1) ASCII, plain-text report file (.txt) identifying the "unbackmasked" sound clips

For the Plus work, four (4) files.

- Two (2) Java archive files (.jar)
  - The archive for the Check Level of this LA
  - A runnable archive with the java.sound.Scale class (entry point)
- One (1) ASCII, plain-text report file (.txt) identifying the "unbackmasked" sound clips
- One (1) sound file. scale.wav

Grading Rubric

Minus - working Backmasking application

Check - working Backmasking application using your data structures

Plus - Check level, plus scale generation application

Back