CSE 332: Data Structures and Parallelism

Exercise 0 - Spec

Exercise 0 Spec (25su)

The objectives of this exercise are

- **Set up your Java environment**
- **Review Java**
- **Gain experience implementing data structures from an ADT**
- **Use Java generics**
- **Use benchmarking to study the running time of algorithms**

Overview

This exercise consists of the following parts:

1. Setting up your programming environment
   - In this section, we will walk you through downloading Java and an IDE (IntelliJ). If you already have Java and an IDE on your machine, you're welcome to use those instead.
   - However, the autograder on Gradescope currently runs on Java 11, so **we strongly recommend you also switch your Java version to Java 11**, as there might be some functionality issues with different versions of Java.
2. Implementing Stacks and Queues
   - In this section, you will implement 4 total data structures, two each for stacks and queues. In each case, one data structure will be LinkedList-like, the other will be ArrayList-like.
3. Benchmarking
   - In this section, we will walk you through the process of benchmarking several implementations of stacks and queues, including your own from the previous part
4. Reflection
   - We will ask you to provide answers to some questions reflecting on what you observed while implementing and benchmarking in the previous two steps.

Code Restrictions

Your implementations in this assignment must follow these guidelines

- You may not add any import statements beyond those that are already present. This course is about learning the mechanics of various data structures so that we know how to use them effectively, choose among them, and modify them as needed. Java has built-in implementations of many data structures that we will discuss for example, don't use ArrayLists.
- Do not have any package declarations in the code that you submit. The Gradescope autograder may not be able to run your code if it does.
- Remove all print statements from your code before submitting. These could interfere with the Gradescope autograder (mostly because printing consumes a lot of computing time)
- Your code will be evaluated by the Gradescope autograder to assess correctness. It will be evaluated by a human to verify the running time. Please write your code to be readable. This means adding comments to your methods and removing unused code (including "commented out" code). For comments, we won't be rigid in grading as in the 12x series. Just make sure your comments are detailed enough. If someone were to read your code for the first time, they would be able to understand what you're trying to accomplish.

Provided Code

Several Java classes have been provided for you in a starter code zip file on the course website. Here is a description of each.

- `MyQueue`
  - A queue interface. It is called `MyQueue` because Java already has a built-in interface called `Queue`, and so this avoids any kind of naming issues
  - Each queue data structure you write should `implement` this interface.
  - *Do not submit this file*
- `LinkedQueue`
  - You will implement a LinkedList-like queue data structure in this class. It is LinkedList-like in the sense that data is stored within node objects, which reference other node objects.
  - ***You will submit this file to Gradescope***
- `ArrayQueue`
  - You will implement an ArrayList-like queue data structure in this class. It is ArrayList-like in the sense that data is stored within an array, which it resizes when it becomes full
  - ***You will submit this file to Gradescope***
- `MyStack`
  - A stack interface. It is called `MyStack` because Java already has a built-in interface called `Stack`, and so this avoids any kind of naming issues
  - Each stack data structure you write should `implement` this interface.
  - *Do not submit this file*
- `LinkedStack`
  - You will implement a LinkedList-like stack data structure in this class. It is LinkedList-like in the sense that data is stored within node objects, which reference other node objects.
  - ***You will submit this file to Gradescope***
- `ArrayStack`
  - You will implement an ArrayList-like stack data structure in this class. It is ArrayList-like in the sense that data is stored within an array, which it resizes when it becomes full
  - ***You will submit this file to Gradescope***
- `NaiveQueue`
  - This class implements the `MyQueue` interface. It is provided as an inefficient implementation of a queue for the purposes of running time comparisons
  - *Do not submit this file*
- `Client`
  - This class performs some basic testing of the above classes. The Gradescope autograder will do more precise testing than this.
  - *Do not submit this file*
- `Benchmark`
  - This class runs and times the above stack/queue implementations

- *Do not submit this file*

Setting Up Your Java Environment

Part 1: Downloading Tools

The first thing you should do is download and install the tools needed to work locally.

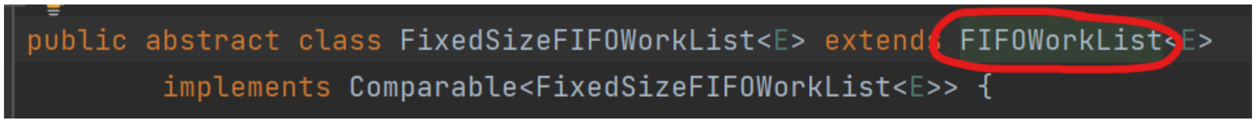
- Download OpenJDK 11 from AdoptOpenJDK.
    - Unless you have an Apple M1-based MacBook (November 2020 and later), download OpenJDK 11 from Azul Systems (direct link). If you do not install this instead, IntelliJ will try to use an Intel x86-64-based JDK running in compatibility mode and will perform significantly slower.
    - Note: You are welcome to use any version of the JDK that you would like; however, there is some risk to this. Due to licensing restrictions, the latest version of Java that Gradescope is permitted to use is 11. If you use a newer version than this, then your environment will differ from our autograding environment, meaning you may have code that works fine on your machine that does not compile or run in Gradescope. This would mostly happen if your code used a method in Java Collections that was added in a later version.
- Download IntelliJ IDEA, our recommended IDE (though you're welcome to use another one if you prefer).
    - If you have an Apple M1-based MacBook, take care to choose the ".dmg (Apple Silicon)" download. It has significant performance implications.
    - As a student, you can get a free education license and can access either the Community or Ultimate Edition of IntelliJ IDEA. If you don't want to register for a JetBrains account, the free Community Edition is totally sufficient for this course.
    - Even if you already have IntelliJ installed, we encourage you to update to the latest version.
- After downloading IntelliJ, make sure to turn off the gen-AI helpers:
    - Make sure to turn off the autocomplete: steps to disable autocomplete
    - If you have AI Assistant installed, make sure to turn it off as well: steps to disable AI Assistant

Part 2: IntelliJ Tips

This is just a compilation of IntelliJ tips some students have found useful, sorted by arbitrary "usefulness":

- Probably one of the most useful features for navigating through our codebase is Ctrl (or Command) clicking on symbols such as class names: https://www.jetbrains.com/help/rider/Navigation_and_Search__Go_to_Declaration.html#eb9663d

  Want to go to `FIFOWorkList` from `FixedSizeFIFOWorkList`?



```
public abstract class FixedSizeFIFOWorkList<E> extends FIFOWorkList<E>
        implements Comparable<FixedSizeFIFOWorkList<E>> {
```

Just Ctrl (or Command) + left-click on `FIFOWorkList` at the top.

- Pressing Shift twice will open the "Search Everywhere" menu that will help you navigate large codebases (such as the projects) easily:
  https://www.jetbrains.com/help/idea/searching-everywhere.html

  Useful for finding that one file you just can't seem to ever find.
- You can refactor (fancy rename) elements of your code such as variables or method names, to help with code organization rather than manually changing everything:
  https://www.jetbrains.com/help/idea/refactoring-source-code.html#refactoring_invoke
- You can search and replace text in the entire repository in IntelliJ:
  https://www.jetbrains.com/help/idea/finding-and-replacing-text-in-project.html
  This is useful for, say, finding an accidentally misplaced `System.out.println` in your project

Implementing Stacks and Queues

This is the main programming portion of this exercise. We will guide you through the process of implementing four different data structures for. You will implement 2 LinkedList style data structures (one for each of stacks and queues) and 2 ArrayList style data structures (again, one for each of stacks and queues). The four data structures are ArrayStack, ArrayQueue, LinkedStack, and LinkedQueue.

	Stack	Queue
ArrayList-Like	ArrayStack	ArrayQueue
LinkedList-Like	LinkedStack	LinkedQueue

For each of the ArrayList-like data structures, the data will be stored in an array. You will use one or more `int` fields to keep track of the indices of the array that contain “valid” items. To add items (i.e., push or enqueue), you will put the new item at some index of the array, and then update the `int` field(s) to indicate the new range of valid items. If the array is not large enough to accommodate the additional item, you must resize the array. Removing an item (i.e., pop or dequeue) requires only updating the `int` fields; in other words, there is **no need to resize when removing an item**. When first initializing your array, you can pick any starting length, though we recommend starting with a smaller value (like 10).

For each of the LinkedList-like data structures, the data will be stored in a `ListNode` object, which has references to other ListNode objects. You will additionally have one or more `ListNode` fields used to access the "chain" of nodes. The `ListNode` class should be a private inner class (i.e., you should not have a standalone `ListNode` class). To add an item (push or enqueue), you will create a new `ListNode` object and update a reference (either a field or another `ListNode`'s reference or both) to include it. Removing an item (pop or dequeue) requires only updating references (either a field or another `ListNode`'s reference or both). *Important: We strongly advise against using doubly-linked nodes (where each node has a next reference and a previous reference) because doubly-linked lists make it twice as hard to keep your references straight.*

For this assignment, there is **no partial credit for passing only a few testcases**. For example, if you failed on any test case on ArrayQueue, you will receive 0 for ArrayQueue. You will receive full credit for ArrayQueue if and only if you passed all tests for ArrayQueue. However, different classes will not affect each other. For example, if you failed one test case on ArrayQueue, you can still get full credit for ArrayStack, but you may not get full credit for ArrayQueue. Each class is worth 10 points on the autograder and 10 points on manual inspection (thus there are 40 possible points on autograder and 40 possible points on manual inspection). Meanwhile, you will be able to see the details on the testcases you failed, and there are no hidden tests on this assignment.

LinkedStack

In `LinkedStack`, you should implement the MyStack interface following the stack ADT. All methods should match the comment describing their behavior in the `MyStack` class. Hint: The main difference between a stack and a queue is that queues add and remove from opposite ends of their data structure, whereas stacks add and remove from the same end. You should ensure that all operations are of **constant time**.

ArrayStack

For `ArrayStack`, implement the `MyStack` interface, following the stack ADT. All methods should match the comment describing their behavior in the `MyStack` class. The same hint is relevant here. Again, make sure that all operations run in **constant time** (except for the case that you must resize when adding an item) and **double your array's length when resizing**.

LinkedQueue

For `LinkedQueue`, you should implement the LinkedList-style data structure that was taught in our first lecture. Review the slides and/or lecture recording on that day to see the details. All methods should match the comment describing their behavior in the `MyQueue` class. Make sure that all operations run in **constant time**.

ArrayQueue

For `ArrayQueue`, you should implement the circular array data structure that was taught in our first lecture. Review the slides and/or lecture recording on that day to see the details. All methods should match the comment describing their behavior in the `MyQueue` class. Make sure that all operations run in **constant time** (except in the case that you must resize when adding an item). **When resizing, you should double the length of the array** (we'll explore why in the Benchmarking section).

Submitting your code

Once you are finished with all of the above, you are finished with the implementation portion of this exercise! To submit your work, you will submit just `LinkedQueue.java, ArrayQueue.java, LinkedStack.java, and ArrayStack.java` to Gradescope under the assignment called "EX00: (Stacks & Queues) Programming Portion". To submit multiple files, you can highlight all four files and drag and drop them into the submission window, or else browse for the files, then select all four. Once you submit, the autograder will test your code for correctness. You can submit infinitely many times as you want before the deadline; thus, we strongly recommend that you modify your code if any tests fail (only completely correct code will receive full credit).

Benchmarking

We recommend that you complete all of the above portions of this assignment before starting on this section, because we will be using your implementations here!

We will guide you through the process of studying the running time of your implementations using benchmarking, which simply means “running the code and seeing how long it takes”. For the majority of the rest of this quarter, we will avoid benchmarking by instead using mathematical definitions of running time. This task in this assignment will help us to 1) see that benchmarking can produce “messy” and hard-to-interpret results, and 2) begin developing an intuition for what is most important and impactful when evaluating running time.

We provide the class `BenchMark` in your starter code for this assignment. There’s a decent amount of code there, but to summarize, here’s what it does.

- By changing the constructor called on line 136 of the Java file in the `csvData` method, you can select which data structure we will benchmark (it’s been set up to work for both a stack and a queue there).
- When you run this code, it will essentially perform many cycles of adding/removing (enqueuing/dequeuing or pushing/popping) items from data structures of various different sizes (ranging from 1 up to approximately 10,000), timing how long it takes to do it. *Tip: When the code is running, DON’T TOUCH ANYTHING. Clicking to another window, skipping a song on your playlist, or interacting with your machine while the test is running may impact your results. If you see a giant spike in the middle of your results, that’s likely what went wrong.*
- After this, it will write the results of the various tests in the `timeTest.csv` file. The format of this will be one line per data structure size. Each line will have two (comma-separated) values. The first value is the size of the data structure, the second is the average time required to add and remove a value 1000 times (in nanoseconds). The format is suitable for copy-pasting directly into a spreadsheet tool (I recommend Google Sheets) for graphing.

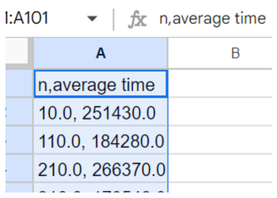
To complete the benchmarking portion of this exercise, first download or copy the Benchmarking Worksheet. You will in your responses on that worksheet, save it as a PDF, then upload that PDF to the “EX00: Benchmarking” Gradescope assignment.

Generating Graphs

The Benchmarking worksheet will ask you to make various graphs demonstrating the running time of your (and some of our) code. Follow these steps to produce a graph.

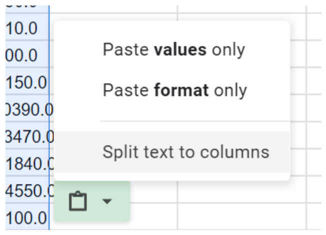
1. After running the main method of `BenchMark.java`, open the `timeTest.csv` file in your IDE. You can then copy the entire contents of the file (Ctrl+A then Ctrl+C).

2. From there, navigate to an empty spreadsheet (the instructions to follow use Google Sheets) and paste the contents there. Your spreadsheet will likely look something like this screenshot:

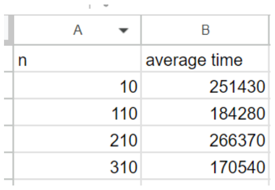


A	B
n,average time	
10.0, 251430.0	
110.0, 184280.0	
210.0, 266370.0	

3. Notice that both numbers on each line are in the same cell. To make our graph, we need to separate the numbers to separate columns. To do that, click the clipboard icon in the bottom-right, then select “Split text to columns”.

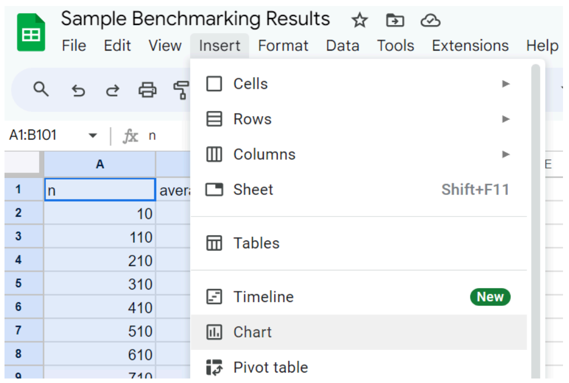


4. Now your spreadsheet should look like this:



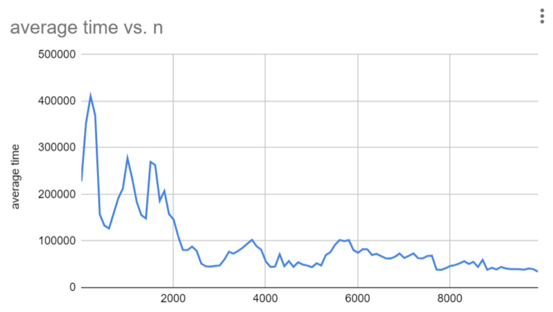
A	B
n	average time
10	251430
110	184280
210	266370
310	170540

5. Next, highlight all of the contents of the spreadsheet and select Insert->Chart

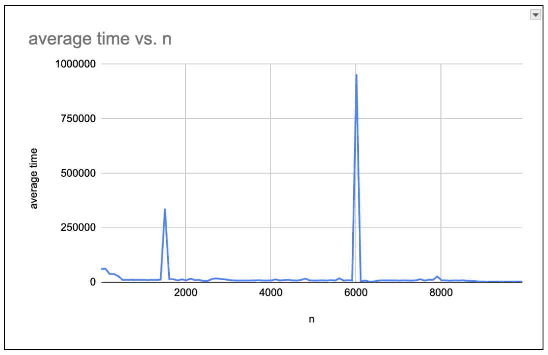

Sample Benchmarking Results
File Edit View Insert Format Data Tools Extensions Help
A1:B101
n
Cells
Rows
Columns
Sheet Shift+F11
Tables
Timeline New
Chart
Pivot table

	A	
1	n	aver
2	10	
3	110	
4	210	
5	310	
6	410	
7	510	
8	610	

6. Now you should automatically have a graph appear to indicate how your running time changes with the size of the data structure! You can take a screenshot of this chart to paste it into the worksheet. My ArrayQueue graph looks like this.



For these graphs, it is okay for them to have a few occasional spikes, like such:



This comes from many things out of our control (Java’s garbage collection, loading a new cache block, etc). Just make sure not to touch your computer while the benchmark file is loading to limit these spikes!