CSE374 – Spring 2021 – Assignment 2

This assignment is done individually. You can search for information online, use your textbook, ask the instructor or Teaching Assistant for help. Working with anyone else is a non-authorized collaboration, that is, an act of academic dishonesty. If you have any doubt or question about what qualifies as academic dishonesty, please contact the instructor to make an informed decision.

Objective: Empirically evaluate and contrast implementations.
Due date: Sunday Feb 22th at 11:59pm. Assignments do not receive a grade after midnight.
How to submit: A ZIP file sent through the course website (no emails accepted).
ZIP your whole Eclipse project, name it "*Firstname_Lastname*_Assignment2.zip"
Submissions in other file formats, *or using other environments*, will not be graded.
You are responsible for ensuring that your submission was submitted. A quick way to check is to download it back from the website and open it.
Grading (14 pts):Each box shows the number of points in a question. Commenting counts for 1 point.

I. Overview

Our lecture focuses on how to evaluate the time complexity and space complexity of an algorithm, using mathematical notations such as ~ and Θ. These are *approximations*, particularly useful to compare algorithms at a design level. To get seconds or bytes, we need to empirically measure what a specific implementation does, on a specific hardware. In this assignment, you will focus on empirically (=experimentally) measuring the time of several algorithms on your computer.

II. Empirically checking whether the new space given to an array matters

0.5 pt

1) Create an ADT named *RemovalArray*, which supports:
   - add(E e)
   - remove(int index)
   - remove(E e)

   These three operations should behave in the same way as in a List (e.g., LinkedList, ArrayList)

1 pt

2) Create an implementation named *IncrementalArray*. It implements the RemovalArray as follows: if the user tries to add an element and there is no more space, then a new array is created with *just one extra space* to accommodate the element. If the user removes an element, all elements after the one removed must be *shifted* to avoid leaving an empty space. For example, if the array was [1 5 2 3 7] and we remove in position 1, then 2-3-7 will shift.

0.5 pt

3) Create an implementation named *DoublingArray*. Its implementation is the same as the IncrementalArray, except that a new array is created with double the size instead of only one more spot. For example, if you have an array of 5 elements and it's full, you will then make an array of 10 elements instead of just 6.

2 pts

4) We now need to create an experimental setup. To perform a measurement, you need **three ingredients**: an operation (here we will add), performed on a lot of data (so that it starts taking a noticeable amount of time on your computer), and performed many times (so that your results capture the *average* processing time).
   • Create a main package, with a Main class. Write a method addToArray which takes two arguments: a RemovalArray, and a number *n* of elements to add (i.e. an integer). This method will add the numbers from 1 to *n* included. For example, if n was 5, then the method would be adding 1, 2, 3, 4, 5 in the array.
   • In your Main class, create a function measureTime that takes the same two arguments: a RemovalArray and a number *n*. It returns *how long it took* to add the n elements to the object. Use the Stopwatch from the textbook (pages 174-175).
   • In your Main class, create a function runExperiments that takes two arguments: a RemovalArray and a filename. It will measure the time necessary to add to that array, by varying n from 20,000 to 400,000 by steps of 20,000. That is, it will call measureTime for

20,000 and then for 40,000 and then for 60,000, all the way to 400,000. Each experiment must be repeated 10 times. Results will be stored in the file whose filename is given as argument. Results must follow this format:

```
N,Repeat1,Repeat2,Repeat3,Repeat4,…
20000,1.04,0.88,1.48,1.06,…
40000,5.04,4.89,5.99,4.99,…
60000,6.66,5.24,8.01,5.20,…
```

Note that these four lines are here to *exemplify* the format. The time data is whimsical. There are more lines than this (you go up to 400,000 included) and more columns (you have 10 repeats). Also note that all values are separated by a comma.

0.5 pt

5) Now that you are *able* to run experiments, let's do this and look at the results! Call runExperiments twice: to get measures for an incremental array (saved in incremental.csv), and for a doubling array (saved in doubling.csv). If the computer is not done fast enough for your taste, you *may* choose to use a System.out.println("*") after every 20,000 so that you can see it's making progress. As long as such personal trackers are not counted by the stopwatch, you are free to include them.
Remember to include your two files of experimental results in your submission.

1.5 pts

6) Create an Excel spreadsheet. To combine your two datasets, simply create one new column named "Implementation". For all lines copied from incremental.csv, the value will be Incremental. For all lines copied from doubling.csv, the value will be Doubling. This way, your experiments are in one place, with a consistent format, and you can tell which one is which.
When you have experiments and variance, you do not only want to show the average. We need to also see the min and max for each experiment. In Excel, you have two options: Box and Whisker, or Stock ("High-Low-Close"). To familiarize yourself with a Box and Whisker, here is a brief description on how to create a simple box and whisker plot:
https://www.excel-easy.com/examples/box-whisker-plot.html
If you prefer, there is a more complete video:
https://www.youtube.com/watch?v=BcwFD0ICOf0
Create two boxplots (one for the incremental array, one for the doubling array).

1 pt

7) Now that you can see the empirical results, you should analyze them. Just creating data or visualizing it is not an analysis: there is no interpretation or insight. Your analysis should attempt to explain why you believe that you are seeing the results as they are.
Include your Excel file as part of your submission. It will include your explanation and your two box plots.

III. Empirically trying a quick remove

1.5 pts

1) In the previous section, your two implementations had a removal by shifting. This can potentially trigger a lot of operations, particularly when you remove elements that are around the beginning of the array. If your array must keep the elements ordered, you don't have many choices. But if the order doesn't matter, you can remove a lot faster.
Write an implementation of RemovalArray named SwappyArray. Its add method should throw an exception (not implemented). We want to implement the remove operations as follows: when an element is removed, the last element in the array is put instead. In other words, you do not shift anything: you simply copy over the last element and make only that spot empty.

0.5 pt

2) You should be testing your code regularly, *even if nobody asks you*. We receive many assignments in which the code doesn't work, and you should have known it such that you could fix it. Here, we'll emphasize the need for testing, while hoping that you'll do it by yourself later on (it's in your own interests!). Write a test method demonstrating that your code works.

1.5 pts

3) Perform empirical measurements to compare the time of a SwappyArray against a DoublingArray in repeatedly removing the first element. Your goal is only to time this remove operation. You should do it on arrays of increasingly large size and repeat your measurements, as you saw in section II. Include your csv outputs and your Excel spreadsheet with plots.

0.5 pt

4) The DoublingArray could have been used to implement a List. But the SwappyArray cannot. Explain why, and state which ADT it would instead be more suitable for.

2 pts

IV. This belongs to a museum

Consider the following code fragment, written in ***antiquated*** (but still common) Java:

```
public static int getSum(List<Integer> L) {
        int total = 0;
        for(int i=0; i<L.size(); i++)
                total+=L.get(i);
        return total;
}
```

We previously saw in our lecture that this style could be a problem *depending* on how the list is implemented. If we use a LinkedList then get(*i*) can be disastrous, because we must go through all elements up to *i*. If we use an ArrayList, then get(*i*) is fast because we directly tap into an array location. We'll use an empirical time analysis to demonstrate this.

Copy/paste the getSum code. Prepare the usual empirical setup, so that you can time the code, write the results to files, and so on. Your goal is to measure how long it takes for getSum to perform on a LinkedList (you'll need to wait a bit) compared to an ArrayList. Your experiments can use lists of size 10,000 up to 200,000 by steps of 10,000.

Your Excel plots should make it clear that getSum with a LinkedList would be a total disaster compared to an ArrayList, all because of a bad code using a get. Include your csv outputs and your Excel spreadsheet with plots.

As part of your Excel, explain how getSum should be written such that performances wouldn't depend on having a LinkedList or ArrayList.

Assignment questions? Doubts? Feel free to contact our teaching assistant, Elinore, at:

eavensek@miamioh.edu