A pile of exams

Problem description

Sometimes when I take an exam off the top of the remaining pile to grade it's all just too much, and I immediately replace it on the bottom of the pile. Sometimes I grade it. But, I never ever add new exams to the pile once I've started. This assignment is about implementing an interface to a data structure that supports these operations, and then making use of it. To keep things simple instead of using "Exams" we'll use `int` as the contents of the pile (think of these as representing either ID numbers, or the indices of a list of exams where the "actual" exams are stored.)

Sometimes I try to grade the exams in order (so that it's easier to enter the grades afterwards). One way to do this is to just search through the pile until I find the first one (moving others to the bottom of the pile), then mark it, then search for the second one, and so on. You'll also be implementing this procedure and investigating its effectiveness.

If I was feeling particularly clever, there is another strategy that I could use when grading exams in order. Rather than blindly moving each exam to the back of the pile while searching for the next one, I could instead scan through the first few and if I find what I'm looking for then take it out and mark it. If I didn't find it then it would make sense to move those that I've scanned through to the bottom of the pile in one operation.

The interface

```
public interface ExamPile {
    public void load(List<Integer> items);
    public int peek();
    public int mark(int depth, int value);
    public void delay(int count);
}
```

Detailed specification

The assignment consists of three parts.

1. Provide a class **EP** which implements the **ExamPile** interface according to the following conditions:

- **load(List<Integer> items)**: Initialises the pile of exams to consist of the contents of the list provided (the elements of the list represent the exams in order from top to bottom of the pile). The internal representation of the pile need not be as a list.
- **size()**: Returns the number of exams remaining in the pile.
- **peek()**: Returns the value at the top of the pile.
- **mark(int depth, int value)**: Returns the given value from the top section of the pile, as specified by depth, and removes it from the pile. Returns -1 if the value is not found.
- **delay(int count)**: Moves *count* values from the top of the pile to the bottom of the pile.

All of **peek()**, **mark()**, and **delay()** should throw an **EmptyPileException** if applied to an empty pile.

2. Suppose that we load a pile with a list consisting of the numbers $0, 1, 2, \dots, n-1$ in some order. An exam pile can be used to output the numbers $0, 1, 2, \dots, n-1$ in that order by delaying until item 0 is on top, then marking it, then delaying until item 1 is on top, then marking it, and so on. Add a method **sortingSteps()** which returns a **String** that represents the steps needed to do this - using characters **D** and **M** to represent delay and marking steps respectively.

   If the original list loaded was **{1,0,2}** then the output of **sortingSteps()** (with depth=1) should be **"DMDMM"**.

3. It is natural to ask, if we count each delay and each marking as a single step, how many steps can we expect it to take to process the pile in the way specified by **sortingSteps()**, or in other words, what is the average length of the string produced by **sortingSteps()** for a random initial permutation. What effect does varying the depth to which we scan through the pile when calling **mark()** have on the number of steps needed to process the pile. Conduct experiments on piles of exams of various sizes and report the results in the form of graphs and/or tables. Write one or two paragraphs analysing the results of the experiments (specifically, how does the average number of steps required scale with the size of the pile, and how is it related to the depth used to scan the pile). No formal justification is required - but ideally you should be able to explain why you think your analysis is likely to be correct.

   Your report should be a one page PDF (single side), containing your analysis and graphs/tables.

Input and output

Your program should read lines of input from *stdin* where each line contains a space separated list of numbers used to load the pile. After each line is loaded your program should print to *stdout* the result of calling **`sortingSteps()`** on the pile, and then the next line is read. Your program should use a default *depth* of 1, which can be altered according to an argument passed on the command line.

So, for example, running your program like this:

```
java week09/EP
```

and giving it this input

```
8 7 6 2 1 0 5 9 3 4

1 8 6 9 0 3 2 7 5 4
```

should result in this output

```
DDDDDMDDDDDDDDDMDDDDDDDMDDMMDDDMDDDMDDMDMM

DDDDMDDDDDMDDDDMDDDDDDDMDDMDDDDMDMDMMM
```

Running your program with a command line argument of 3 on the same input:

```
java week09/EP 3
```

should result in this output

```
DMMMMMMDMMMM

DMDDMDMMMMMMMM
```

There are various methods you could use to generate the data needed for part 3. For instance, you might write a separate program (which you do not need to submit) that generates random piles of exams (of various sizes) and runs sortingSteps on them (to various depths). Alternatively, if your original program were to write the length of the sortingSteps() string to stderr, then you could use it on a data file you create that consists of various random piles of exams.

Group work and Submission

For this assignment we require you to work in teams of two people. You may select your own group and inform us of your choice by 4pm Tuesday April 19th.

Send an email to ihewson@cs.otago.ac.nz letting us know the University user code of both students in your team. Any students who don't select their own team will be assigned to one by us and informed via email. Teams will be assigned in a pseudo-random manner. Students will be paired to others who have completed a similar amount of internal assessment.

By week 8 of the semester, you will have been notified by email how to check your assignment against some simple test files to make sure that it is basically working correctly.

Your assignment code should be in the package **`week09`**. You will be able to submit your assignment using the command:

```
asgn-submit
```

Any assignments submitted after the due date and time will lose marks at a rate of 10% per day late.

Both students in each pair should submit their work (which should be identical) using **`asgn-submit`**. When submitting your assignment you will be asked to rate the contribution of you and your team mate using the scale:

1. I did none of the work. My team mate did it all.
2. I did a little. My team mate did most of the work.
3. We contributed evenly.
4. I did most of the work. My team mate did a little.
5. I did it all. My team mate did none of the work.

Do *not* work with any other student who is not in your pair, or ask the demonstrators for help with the assignment. However, feel free to come and discuss any questions you may have with Iain Hewson.

All submissions will be checked for similarity.

Marking

This assignment is worth 10% of your final mark for Cosc 241. It is possible to get full marks. In order to do this you must write correct, well commented code which meets the specifications, and a sensible, well presented report.

Marks are awarded for your program based on both implementation (4%) and style (3%), and for your report (3%). It should be noted however that it is very bad to style to have an implementation that doesn't work.

In order to maximise your marks please take note of the following points:

- Your code should compile without errors or warnings.
- Your program should use good Java layout (use the **`checkstyle`** tool to check for layout problems).
- Avoid duplicated code. Strive for conciseness, but not at the expense of clarity.
- Make sure each file is clearly commented.
- Most of your comments should be in your method headers. A method header should include:
  - A description of what the method does.
  - A description of all the parameters passed to it.
  - A description of the return value if there is one.
  - Any special notes.

Part of this assignment involves you clarifying exactly what your program is required to do. Don't make assumptions, only to find out that they were incorrect when your assignment gets marked.

If you have any questions about this assignment, or the way it will be assessed, please see Iain or send an email to ihewson@cs.otago.ac.nz.