Lab #1

***CSE2331/5331* (*Spring 2013*)**

Due Date: Monday, February 11 by 11:59pm

Guidelines

- Any lab that does not compile will receive a **ZERO**. The grader will not do this for you when grading.
- All make-ups for lab must be accompanied by a documented and verifiable excuse well before the deadline. Given the severity of the emergency please inform me as soon as possible.
- Lab submissions will NOT be accepted via email to me or the grader.
- Work on the lab on your own, i.e. individually. **No group work**!
- ***All suspected cases of academic misconduct will be reported to the University Committee on Academic Misconduct for review.***

Objectives

Develop and analyze a sorting algorithm using the Stack data structure.

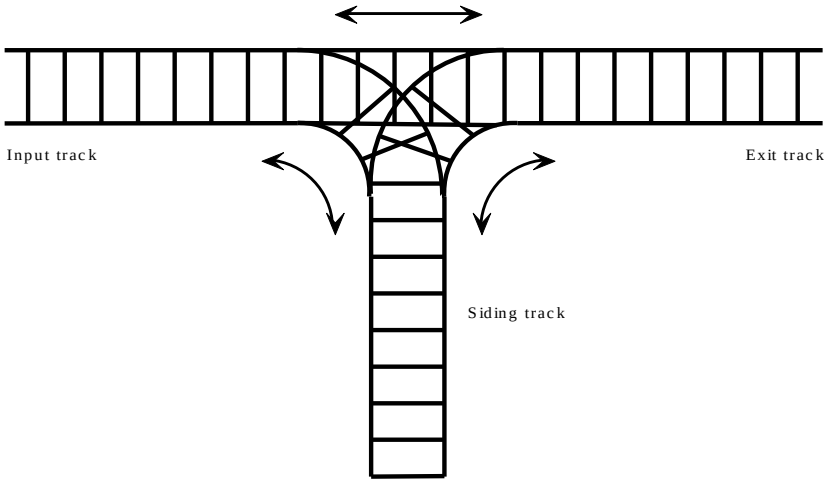
Materials

Implement your solution using either *Java* or *C++*. The grader will grade *Java* solutions using the *Eclipse IDE* for *Java* (I highly recommend this choice) installed on our *Window*s lab computers and grade *C++* solutions using the GNU *g++* compiler installed on our *stdlinux* system. *If you develop your code at home using a different IDE or compiler than what is available on our systems, I highly suggest you port your code to one of the platforms the grader will use and ensure your code compiles and runs properly before you submit***.** You can download the Eclipse IDE from www.eclipse.org for free.

I provide initial code in which you will insert your solution. The grader will use the methods(functions) I have provided to grade your work. For those using *Java*, I have provided JUnit test code. *C++* coders can take a look at all the *Java* code I provide to get an idea on how to time and test your code in a similar fashion. Please find on Carmen the following files: Java - ***Lab1.java, Lab1Test.java, RandomInputGenerator.java,*** and ***StopWatch.java***; C++ - ***Lab1.h*** and ***Lab1.cpp***.

Coding

Write a program that sorts cars on a train. Consider the following diagram of a train track.



Cars come in from the left and exit on the right hand side. They can also move onto the "siding" track to the bottom. At the intersection, each car on the input train can either move to the exit track or onto the siding. From the siding, each car can either move to the exit track or back onto the input track. And finally, from the exit track, a car can move back to the input or siding tracks. The input, exit, and siding tracks are stacks, since you can only access the car at the head of the track (as viewed from the intersection point). Consider that all tracks are at least as long as the input train.

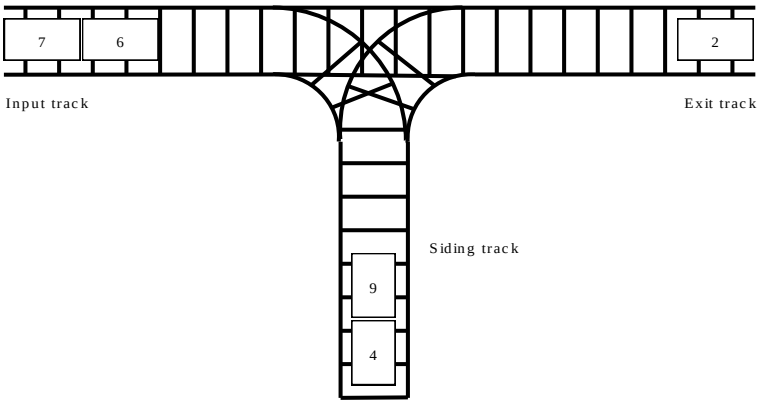
The input to your program will be a sequence of car numbers (input track). Your output is the exit track (sorted as an ascending sequence) and a sequence of instructions to a railroad worker that will sort the cars. Here is an example input and output:

Input track: `7 6 2 9 4`
Output track: `2 4 6 7 9`

Output to the screen:

```
1.  Move 4 from input to siding
2.  Move 9 from input to siding
3.  Move 2 from input to exit
4.  Move 9 from siding to input
5.  Move 4 from siding to exit
6.  Move 9 from input to siding
7.  Move 6 from input to exit
8.  Move 7 from input to exit
9.  Move 9 from siding to exit
```

After moves 1 through 3, the train will look like the following:



Your solution must conform to the following constraints (***Note, not adhering to all of these criteria exactly will result in a drastic reduction in your grade!***):

- Choose either the programming language *Java* or *C++* for your implementation.
- Do not change the method (function) signature for ***trainSort***. The grader's test code will call this method(function) exactly as I have defined it.
- Insert your solution into the ***trainSort*** method(function) inside the Lab1 Class. You may define additional *helper methods*, but define these to be *private* and *static* inside the Lab1 Class.
- You must report to the terminal/Console only moves between the three stacks (i.e., input, siding, and exit).
- Your solution must use the **stack** classes as defined in the *Java API* or the *Standard Template Library* (STL) in *C++* to represent the three train tracks in solving this problem.
    - Though, you are only allowed to use the following methods(functions) of the stack class:
        - *Java – empty*, *peek*, *push*, and *pop*. See the *Java API* - http://docs.oracle.com/javase/7/docs/api/.
        - *C++ - empty*, *top*, *push*, *pop*.

      * You are NOT allowed to use methods(functions) not listed here, such as *search*, *size*, *get*, etc, which include methods(functions) from classes higher up in the class hierarchy to the stack class.
      * You are also NOT allowed to use iterators.
    - You may use as many stacks as you think are needed in your solution.
- No global variables! All variables you declare must have local scope. Thus, no instance variables are needed for the Lab1 class.
- The content and ordering of the items within each of the three stacks (input, siding, or exit) cannot be destroyed by your algorithm and is determined only by moving items between these three stacks. In other words, let's say your solution decides to move an item between the three stacks and then follows this with yet another move between the three stacks. Before your solution performs the latter move, you should ensure that all three stacks have not had any of their items re-arranged nor removed since the completion of the prior move. For example, let's say that currently the siding stack has the following values: 7 6 4 2, which were added to it previously from either the input or exit stacks. Your algorithm then

damages the siding track by removing the 2 and 4 onto an alternate stack. Before your solution can move any items between the three stacks, you must ensure that the siding track is back to its undamaged state, i.e. it has all the values 7 6 4 2 in the correct order.

- Document your code well. This means put your **name**, **date**, and **brief description of the problem** you are solving at the top. Provide multi-line comments for each method(function) and then single line comments in your code where appropriate (You do NOT need a comment for almost every line of code!). Provide single line comments for sections of code that represent a discernible algorithm. You will receive point reduction for too few or too many comments.

Analysis

- Include all of the following in your report:
  - Describe what input would result in the worst case timing of your algorithm.
  - Describe what input would result in the best case timing of your algorithm.
  - Algorithm analysis
    - Time your algorithm using ***asymptotic notation***. Note, simply stating a final timing will receive NO credit. Show in detail (even line by line next to your program's listing) how you arrive at your timing(s). No hand waving please. The grader will pay particular attention to the reasons behind your statements.
  - Empirical Algorithm Analysis
    - Run your solution on various size input and provide one or more graphs that plot input size vs. run-time (say in milliseconds). Use enough runs of your code to have your graph(s) illustrate something substantial, interesting, and correct.
    - Describe in your report whether the asymptotic and empirical analyses conform to each other or differ. In either case, clearly **discuss** your observations and findings.

Grading

- Correctness of the implementation (85%) – Code passes all tests I have provided and of course any you dream up. The grader has another collection of tests that your code will be run against.
- Documentation and code readability (5%) – This includes good indentation and descriptive variable and function names.
- Report(10%) – This is a lab report and NOT just a cut-and-paste of numbers and graphs (just these and little to no discussion will receive little to no credit!). Yes, your report must be a discussion. It must include the following sections: Introduction and problem description, motivation, algorithmic presentation of your solution, and clear presentation of your analysis results. You should use a

word processing program, e.g. MS Word. Please save the final report file in PDF format.

Submission

Create a single zip file with the following work: 1) README file that includes how to compile and run your source code, 2) Well documented source code (Do NOT include executables or any compiled code), 3) Code used for generating your analysis results, and 4) Report.

Submit the zip file to the Carmen dropbox.