COMP2402: Data Structures

Assignment 2: Min-Stacks and Min-Deques

Grading

This assignment will be tested and graded by a computer program (and you can submit as many times as you like). For this to work, there are some important rules you must follow:

- Keep the directory structure of the provided zip file. If you find a file in the subdirectory `comp2402a2` leave it there.
- Keep the package structure of the provided zip file. If you find a `package comp2402a2;` directive at the top of a file, leave it there.
- Do not rename or change the visibility of any methods already present. If a method or class is `public` leave it that way.
- Submit early and often. The submission server compiles and runs your code, and gives you a mark. You can submit as often as you like and only your best submission will count. There is no excuse for submitting code that does not compile or does not pass tests.
- Write efficient code. The submission server places a limit on how much time it will spend executing your code, even on inputs with a million lines. For some questions it also places a limit on how much memory your code can use. If you choose and use your data structures correctly, your code will easily execute within the time limit. Choose the wrong data structure, or use it the wrong way, and your code will be too slow for the submission server to grade (resulting in a grade of 0).

Submitting and Testing

The submission server is now accepting submissions. Please try it out.

Warning: Do not wait until the last minute to submit your assignment. There is a hard 5 second limit on the time each test has to complete. For the largest tests, even an optimal implementation takes 3 seconds, and may take longer if the server is heavily loaded.

The Assignment

Start by downloading the Assignment 2 Zip File, which contains a skeleton of the code you need to write.

The `Tester` class, included in the zip file gives a very basic demonstration of the code. As is, `Tester` throws an exception when it tries to test `FastMinStack` and `FastMinDeque` because they're not implemented yet. Despite the name, `Tester` does not do thorough testing. The submission server will do thorough testing.

1. [5 marks] A `MinStack` supports three main operations: the standard `Stack` operations `push(x)` and `pop()` and the non-standard `min()` operation which returns the minimum value stored on the stack. The zip file gives an implementation `SlowMinStack` that implements these operations so that `push(x)` and `pop()` each run in $O(1)$ time, but `min()` runs in $\Theta(n)$ time.
   For this question, you should complete the implementation of `FastMinStack` that implements all three operations in $O(1)$ time per operation. As part of your implementation, you may use any of the classes in the Java Collections Framework and you may use any of the source code provided with the Java version of the textbook. Don't forget to also implement the `size()` and `iterator()`* methods.
   Think carefully about your solution before you start coding. Here are two hints: (i) don't use any kind of `SortedSet` or `SortedMap`, These all require $\Omega(\log n)$ time per operation; (ii) take a look at the sample solution for Part 10 of Assignment 1. *Really*

understanding this solution will help you design your data structure.

2. [5 marks] A `MinDeque` supports five operations: The standard `Deque` operations `addFirst(x)`, `removeFirst()`, `addLast(x)`, and `removeLast()` and the non-standard `min()` operation that returns the minimum value stored in the `Deque`. Again, the zip file provides an implementation `SlowMinDeque` that supports each of the first four operation in $O(1)$ time per operation but requires $\Omega(n)$ time for the `min()` operation.

   For this question, you should complete the implementation of `FastMinDeque` that implements all five operations in $O(1)$ (amortized) time per operation. As part of your implementation, you may use any of the classes in the Java Collections Framework and you may use any of the source code provided with the Java version of the textbook.

   Don't forget to also implement the `size()` and `iterator()`* methods. Think carefully about your solution before you start coding. Here are two hints: (i) don't use any kind of `SortedSet` or `SortedMap`, These all require $\Omega(\log n)$ time per operation; (ii) consider using one of the techniques we've seen in class for implementing the `Deque` interface.

*You can still get part marks without correctly implementing the `iterator()` method. But for full marks, the `iterator()` method needs to return an `Object` that supports the `next()` and `hasNext()` methods from the `Iterator<T>` interface. For a `MinStack`, the iterator should output the elements at the bottom of the stack first (the elements that have been in the stack the longest), finishing with the element at the top of the stack (the element that would be returned by a call to `pop()`). For a `MinDeque`, the iterator should output the elements beginning with the first element (the one that would be returned by `removeFront()`) and finishing with the last element (the one that would be returned by `removeLast()`).

Hint: There's no need to build your iterators from scratch. The collections in `java.util` all have iterators.