COMP 2402 AB- Fall 2023 Assignment #2

Due: Wednesday, October 11, 23:59

Submit early and often. Late submissions (up to 12 hours) will be accepted.

Academic Integrity

You may:

- Discuss general approaches with course staff and your classmates,
- Use code and/or ideas from the textbook,
- Use a search engine / the internet to look up basic Java syntax.

You may not:

- Send or otherwise share code or code snippets with classmates,
- Use code not written by you, unless it is code from the textbook (and you should cite it in comments),
- Use a search engine / the internet to look up approaches to the assignment,
- Use code from previous iterations of the course, unless it was solely written by you,
- Use the internet to find source code or videos that give solutions to the assignment.

If you ever have any questions about what is or is not allowable regarding academic integrity, please do not hesitate to reach out to course staff. We will be happy to answer. Sometimes it is difficult to determine the exact line, but if you cross it the punishment is severe and out of our hands. Any student caught violating academic integrity, whether intentionally or not, will be reported to the Dean and be penalized. Please see Carleton University's Academic Integrity page.

Grading

This assignment will be tested and graded by a computer program (**you can submit it many times; your highest grade is recorded**). 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 modify any of the provided interfaces.

- Do not rename/delete any of the supplied files.
- 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

For every assignment on Brightspace, you will find a URL "Assignment # submission server" (replace # with the corresponding assignment number) that will take you to the submission page. For more details on how to submit your work, follow the instructions on the document "Submission instructions". If you have issues, please post to Discord to the teaching team (or the class) and we'll see if we can help.

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 full 4 seconds.

Start by downloading and decompressing the Assignment 2 Zip File (comp2402a2.zip), which contains a skeleton of the code you need to write. The skeleton code in the **zip** file compiles fine. Here's what it looks like when you unzip and compile it from the command line:

```
alina@euclid:~$ unzip comp2402a2.zip
Archive:  comp2402a2.zip
  inflating: comp2402a2/SuperStack.java
  inflating: comp2402a2/SuperSlow.java
  inflating: comp2402a2/SuperFast.java
  inflating: comp2402a2/DuperDeque.java
  inflating: comp2402a2/DuperSlow.java
  inflating: comp2402a2/DuperFast.java
  inflating: comp2402a2/Tester.java
alina@euclid:~$ javac comp2402a2/*.java
```

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 `SuperFast` and `DuperFast` because they're not implemented yet. Despite the name, `Tester` does not do thorough testing. The submission server will do thorough testing. To run the `Tester` from the command line:

```
alina@euclid:~$ java comp2402a2.Tester
```

The Assignment

This assignment contains two main parts:

Part 1 [50 marks]: A `SuperStack` is an extended stack that supports four main operations: the standard Stack operations `push(x)` and `pop()` and the following non-standard operations:

- `max()`: returns the maximum value stored on the Stack.
- `ksum(k)`: returns the sum of the top k elements on the Stack.

The zip file gives an implementation `SuperSlow` that implements these operations so that `push(x)` and `pop()` each run in $O(1)$ time, but `max()` and `ksum(k)` run in $O(n)$ time. For this question, you should complete the implementation of `SuperFast` that implements all four 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:

balance method

1. don't use any kind of `SortedSet` or `SortedMap`, these all require $\Omega(\log n)$ time per operation.
2. think about how the maximum on the stack changes as new elements are pushed. Understanding this will help you design your data structure.

Part 2 [50 marks]: A `DuperDeque` is an extended Deque that supports seven operations: The standard Deque operations `addFirst(x)`, `removeFirst()`, `addLast(x)`, and `removeLast()` and the following non-standard operations:

- `max()`: returns the maximum value stored on the Deque.
- `ksumFirst(k)`: returns the sum of the first k elements on the Deque.
- `ksumLast(k)`: returns the sum of the last k elements on the Deque.

use solution from first problem

Again, the zip file provides an implementation `DuperSlow` that supports each of `addLast(x)` and `removeLast()` operations in $O(1)$ time per operation but requires $\Omega(n)$ time for the other operations.

maintain 3 : 1 ratio

For this question, you should complete the implementation of `DuperFast` that implements all seven 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:

balance()

1. don't use any kind of `SortedSet` or `SortedMap`, these all require $\Omega(\log n)$ time per operation;
2. you can write additional functions to support your design choices; consider using one of the techniques we've seen in class for implementing the Deque interface.

Tips, Tricks, and FAQs

How should I approach each problem?

- Make sure you understand it. Construct **small** examples, and compute (by hand) the expected output. If you aren't sure what the output should be, go no further until you get clarification.
- Now that you understand what you are supposed to output, and you've been able to solve it by hand, think about how you solved it and whether you could explain it to someone. How about explaining it to a computer?
- If it still seems challenging, what about a simpler case? Can you solve a similar or simplified problem? Maybe a special case? If you were allowed to make certain assumptions, could you do it then? Try constructing your code incrementally, solving the smaller or simpler problems, then, only expanding scope once you're sure your simplified problems are solved.

How should I test my code?

- You can modify the `Tester` class. For example you can change the "20" in `superTest(new SuperFast(), 20);` to a smaller/bigger number to test less/more operations.
- The `Tester` class provided with the assignment does a sequence of adds, followed by a sequence of removes. This is a very basic test and far from an exhaustive one. It is strongly recommended to modify the tester and design your own test cases. For starters, you can interleave the add and remove operations. Take it even further by making the operation sequence entirely random. This will help you to discover the weaknesses in your solution.
- You should be testing your code as you go along.
- Beware of integer overflow. Note that `ksum` is of type `long` - not `int`.
- Think about tricky cases. For instance, how do the operations behave when the stack/deque is empty, or `k > n` or `k <= 0`? You can always refer to the slow implementations to answer these questions. Although they are slow, they are correct implementations. They can also be useful if you want to test your implementation against a reference one.
- `int` and `Integer` are not the same. Do not use them interchangeably.
- Use small tests first so that you can compute the correct solution by hand.
- Test for speed.