Data Structure and Programming Spring 2025
Programming Assignment 1

Chiao-Yun, Chin b10901154@ntu.edu.tw
Yi-Chen, Wang b10901143@ntu.edu.tw
Yi-Chou, Lin b10901149@ntu.edu.tw

Release Date: 09:00 on 03/13/2025
Due Date: 23:59 on 03/27/2025
Late policy: Score = Score $\times 0.7^{(h/24)}$, h = late hours ≤ 48

1 Problem Statement

This programming assignment asks you to read in a series of commands e.x. (`PUSH 10`, `PUSH 9`, `POP`, ...), and execute them using a stack and a queue separately. Additionally, you should implement a queue using two stacks. We have written code templates for you, and you should only fill in the `TODO` sections in `stack_queue.py`.
Notice:

- Basic Requirements: Implement stack, queue, and queue_by_stack in `stack_queue.py`.
- Implementation Constraints: Modify only `TODO` sections, call the class `node()` when storing your data structure
- Performance Requirements: O(1) operations, amortized O(1) for queue_by_stack (If you are not familiar with amortized analysis, please refer to the appendix)
- Please use Python 3.10 to run `stack_queue.py`.
- Don't use Windows, use Linux instead.

2 Input/Output Specification

2.1 Input Format

Inputs are a series of commands separated by newlines. Below is an example:

```
PUSH 10
PUSH 9
POP
PUSH 8
PUSH 7
PUSH 6
POP
POP
```

Figure 1: Input format

2.2 Output Format

The output format for stack, queue, and queue_by_stack has been fully provided in the code templates. No modifications to the output format are needed.
Below is an example of stack state after each operation of Figure 1:

```
>>Node10
>>Node10>>Node9
>>Node10
>>Node10>>Node8
>>Node10>>Node8>>Node7
>>Node10>>Node8>>Node7>>Node6
>>Node10>>Node8>>Node7
>>Node10>>Node8
```

Figure 2: Output format for stack

Each operation should update and display the current state of the stack or queue. The format follows these rules:

- The stack is displayed in LIFO (Last-In, First-Out) order, where the topmost element is on the right.
- The queue follows FIFO (First-In, First-Out) order, meaning the earliest pushed element is removed first.
- The queue_by_stack follows FIFO order as well. It is displayed by two stacks separately, where the first stack is the front and the second stack is the rear.
- The output format for both structures is the same, with each node represented as **>>NodeX**, where **X** is the value stored in the node.

Below is an example of queue state after each operation of Figure 1:

```
>>Node10
>>Node10>>Node9
>>Node9
>>Node9>>Node8
>>Node9>>Node8>>Node7
>>Node9>>Node8>>Node7>>Node6
>>Node8>>Node7>>Node6
>>Node7>>Node6
```

Figure 3: Output format for queue

Below is an example of queue_by_stack state after each operation of Figure 1:

Stack1: , Stack2: >>Node10
Stack1: , Stack2: >>Node10>>Node9
Stack1: >>Node9, Stack2:
Stack1: >>Node9, Stack2: >>Node8
Stack1: >>Node9, Stack2: >>Node8>>Node7
Stack1: >>Node9, Stack2: >>Node8>>Node7>>Node6
Stack1: , Stack2: >>Node8>>Node7>>Node6
Stack1: >>Node6>>Node7, Stack2:

Figure 4: Output format for queue_by_stack

3 Submission

Submit `stack_queue.py` in a directory named `studentID` and compress it into `studentID.zip`. Finally, upload `studentID.zip` to NTU COOL. The deadline is 3/27 at 23:59.

```
b10901143.zip/
    b10901143/
        stack_queue.py
```

4 Evaluation

Your code will be evaluated on five test cases. Three test cases are provided: `input1.txt`, `input2.txt`, and `input3.txt`. Each test case corresponds to three different golden outputs: `golden_stack_X.txt` (stack implementation), `golden_queue_X.txt` (queue implementation), and `golden_queue_by_stack_X.txt` (queue implemented using two stacks), where `X` represents the corresponding test case number.
A script, `evaluation.sh`, is provided to check whether your code passes these test cases and to measure runtime.
Each test case (20%) consists of three parts:

- **Stack** (8%)
- **Queue** (8%)
- **Queue_By_Stack** (4%)

Passing each one with amortized O(1) time complexity earns the corresponding percentage of the total score.

5 Appendix

Amortized analysis focuses on computing the **average cost** of a sequence of operations. Even if some operations occasionally take O(n) time, their impact is distributed over multiple operations, making the average cost per operation O(1).
For example, consider a data structure where an expensive O(n) operation occurs only occasionally, while most operations take O(1) time. If this O(n) operation happens once every n operations, the total time complexity for n operations is:

$$O(n) + n \times O(1) = O(n) + O(n) = O(n)$$

The average time per operation is:

$$\frac{O(n)}{n} = O(1)$$