*“It’s hard enough to find an error in your code when you’re looking for it; it's even harder when you’ve assumed your code is error-free.”*

*- Steve McConnell*

Dear reader, I welcome you to the second problem of the ongoing series of four problems where you need to implement a queue using stacks and inversely, a stack using queues. I request to solve the following 4 problems in consecutive fashion.

1. **Queue to Stack Adapter**
   1. [**Queue to Stack Adapter - Push Efficient**](https://www.pepcoding.com/resources/online-java-foundation/stacks-and-queues/queue-to-stack-adapter-push-efficient-official/ojquestion)
   2. [**Queue to Stack Adapter - Pop Efficient**](https://www.pepcoding.com/resources/online-java-foundation/stacks-and-queues/queue-to-stack-adapter-pop-efficient/ojquestion)
2. **Stack to Queue Adapter**
   1. [**Stack to Queue Adapter - Push Efficient**](https://www.pepcoding.com/resources/online-java-foundation/stacks-and-queues/stack-to-queue-adapter-add-efficient-official/ojquestion)
   2. [**Stack to Queue Adapter - Pop Efficient**](https://www.pepcoding.com/resources/online-java-foundation/stacks-and-queues/stack-to-queue-adapter-remove-efficient-official/ojquestion)

We are currently solving the part *Queue to Stack Adapter - Pop Efficient*.

***Problem Statement***

* You are required to complete the code of the QueueToStackAdapter class we have provided.
* You have only 2 queue data members available- *mainQ* and *helperQ*. (mainQ is to contain data and helperQ is to assist in operations).
* You need to implement a stack using these 2 queues with all the operations of queues.
* You need to complete the code of following operations of stack:
  + ***push***: Should accept new data in LIFO manner.
  + ***pop***: Should remove and return data in LIFO manner. If not available, print “*Stack underflow*" and return -1.
  + ***top***: Should return data in LIFO manner. If no element is available, print "*Stack underflow*" and return -1.
  + ***size***: Should return the number of elements available in the stack.

*Note*: Please do not declare any stack data structure, you must solve this problem by only using the two instances of queue available.

In this ‘*Pop* *Efficient*’ variation, pop operation should be as efficient as possible (O(1) per call). In order to achieve ***constant time pop (& top) operations***, you can take linear time in push operation.

If you are unable to understand the problem, then you can watch the [*question video*](https://www.youtube.com/watch?v=4uHZxkUw-F4) for better understanding.

***Solution - (O(n) Push, O(1) Pop)***

Let us look at the operations:

* ***pop***: Pop operation should be O(1), hence we will try to maintain the last element at the front of the main queue, by managing push operation somehow.

So, when the main queue is empty, then print “Stack underflow” and return -1. Else, since the last element is brought to the front of the main queue mainQ (somehow we will manage to do this), we will dequeue this element from the main queue using *mainQ.remove()*, and return the element.

* ***top***: Top operation is very similar to pop operation, hence it is also constant O(1) operation in this case.

When the main queue is empty, then print “Stack underflow” and return -1. Else, since the last element is brought to the front of the main queue mainQ (somehow we will manage to do this), we will return this front element using *mainQ.peek()*.

* ***Size***: Size function is also simple, just return the size of main queue mainQ

Now, let us look at what we should do in the push function, so that we can implement LIFO order (stacks) using the FIFO order in the two queues.

So, we need that whenever we would call pop operation, we should have the element which was inserted at last to be at the front of the main queue, so that pop operation is O(1).

But, the queue can enqueue elements only at the back (rear end). So, we will have to take the help of the auxiliary/helper queue provided to us.

Whenever we have to push an add element into the main queue, then first we will dequeue all the elements, one by one from the main queue and enqueue them to the helper queue. Now, the main queue is empty and thus we can add the element to the main queue, which will get to the front (since there is no other element).

Now, we will do the reverse process, i.e. dequeue the elements from the helper queue, one by one and enqueue them back to the main queue.

If I have to give you a summary of push operation, then we are maintaining the elements in the main queue in reverse order, i.e. the element inserted at last will be at front and element inserted in starting will be present in the rear end (at last).

Please refer to the [solution video](https://www.youtube.com/watch?v=bJpPuLMiUgA) if you find difficulty in understanding the solution completely.

***Pseudo Code/ Algorithm***

1. ***Push***:
   1. Dequeue all elements one by one from the main queue and enqueue them into the helper queue.
   2. Add the current element to be pushed into the main queue.
   3. Dequeue all elements one by one from the helper queue and enqueue them back into the main queue.
2. ***Size***: Return the size of main queue using *mainQ.size()*
3. ***Top***: If the main queue is empty, print “Stack underflow” and return -1. Else return the front element of the main queue using mainQ.peek().
4. ***Pop***: If the main queue is empty, print “Stack underflow” and return -1. Else, return the front element of the main queue using mainQ.remove(). (remove operation of mainQ will dequeue the front element as well).

***Implementation (Java)***

How about first trying by yourself without reading the code we provide?

import java.io.\*;

import java.util.\*;

public class Main {

public static class QueueToStackAdapter {

Queue<Integer> mainQ;

Queue<Integer> helperQ;

public QueueToStackAdapter() {

mainQ = new ArrayDeque<>();

helperQ = new ArrayDeque<>();

}

int size() {

return mainQ.size();

}

void push(int val) {

while(mainQ.size() > 0){

helperQ.add(mainQ.remove());

}

mainQ.add(val);

while(helperQ.size() > 0){

mainQ.add(helperQ.remove());

}

}

int pop() {

if (size() == 0) {

System.out.println("Stack underflow");

return -1;

} else {

return mainQ.remove();

}

}

int top() {

if (size() == 0) {

System.out.println("Stack underflow");

return -1;

} else {

return mainQ.peek();

}

}

}

public static void main(String[] args) throws Exception {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

QueueToStackAdapter st = new QueueToStackAdapter();

String str = br.readLine();

while (str.equals("quit") == false) {

if (str.startsWith("push")) {

int val = Integer.parseInt(str.split(" ")[1]);

st.push(val);

} else if (str.startsWith("pop")) {

int val = st.pop();

if (val != -1) {

System.out.println(val);

}

} else if (str.startsWith("top")) {

int val = st.top();

if (val != -1) {

System.out.println(val);

}

} else if (str.startsWith("size")) {

System.out.println(st.size());

}

str = br.readLine();

}

}

}

This code is written and explained by our team in [this video](https://www.youtube.com/watch?v=bJpPuLMiUgA). Please refer to it if you are stuck somewhere.

***Time & Space Complexity Analysis***

Now, it should be simple to analyze!

***Push - O(n)***: We are first removing all elements from main queue and adding them to helper queue, which is n \* O(1) = O(n), then add the current element to be pushed to main queue, which is O(1), then add elements back from the helper queue by removing them one by one, which is n \* O(1) = O(n) again.

Hence, the total time complexity will be O(n) per call, where n is the number of elements already present in the queue.

***Size - O(1)***: We are just returning *mainQ.size()*.

***Pop - O(1)***: Checking if the queue has 0 size or not, and then popping the front element(using mainQ.remove()) are O(1) constant operations.

***Top - O(1)***: Checking if the queue has 0 size or not, and then returning the front element (using mainQ.peek()) are O(1) constant operations.

I hope you enjoyed solving the problem with me. We will come with the next pair of the problems ‘Stack to Queue Adapter’ (Push Efficient and Pop Efficient) where you need to implement queue data structure using only stacks. Until then *keep coding*!

Contributor : [Archit Aggarwal](https://www.linkedin.com/in/archit-aggarwal-6a7716189/)