**1. Problem Discussion**

I request to solve the following 4 problems in consecutive fashion:

[**Queue to Stack Adapter - Push Efficient**](https://nados.pepcoding.com/content/eb9863ac-63ac-4b94-881f-0aeb24df0985/0c54b191-7b99-4f2c-acb3-e7f2ec748b2a/ae41ae29-11ca-4ae4-8ec7-72a411fd59b6/60fb381d-df57-448f-a4a4-8b368f1dffee/9847c2b3-e3ad-4b1c-97d1-00206b1be68d/question/59efbdad-3a31-495a-b7fb-1b4b45ca499f)

[**Queue to Stack Adapter - Pop Efficient**](https://nados.pepcoding.com/content/eb9863ac-63ac-4b94-881f-0aeb24df0985/0c54b191-7b99-4f2c-acb3-e7f2ec748b2a/ae41ae29-11ca-4ae4-8ec7-72a411fd59b6/60fb381d-df57-448f-a4a4-8b368f1dffee/9847c2b3-e3ad-4b1c-97d1-00206b1be68d/question/5d8266e1-96e7-402d-b115-6130f3ceb1f9)

[**Stack to Queue Adapter - Add Efficient**](https://nados.pepcoding.com/content/eb9863ac-63ac-4b94-881f-0aeb24df0985/0c54b191-7b99-4f2c-acb3-e7f2ec748b2a/ae41ae29-11ca-4ae4-8ec7-72a411fd59b6/60fb381d-df57-448f-a4a4-8b368f1dffee/9847c2b3-e3ad-4b1c-97d1-00206b1be68d/question/a6a7b0b2-8f1b-4dd2-8b55-62c5a3bb595f)

[**Stack to Queue Adapter - Remove Efficient**](https://nados.pepcoding.com/content/eb9863ac-63ac-4b94-881f-0aeb24df0985/0c54b191-7b99-4f2c-acb3-e7f2ec748b2a/ae41ae29-11ca-4ae4-8ec7-72a411fd59b6/60fb381d-df57-448f-a4a4-8b368f1dffee/9847c2b3-e3ad-4b1c-97d1-00206b1be68d/question/c6d52185-37e5-4c81-9265-708787285124)

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 'Push Efficient' variation, push operation should be as efficient as possible (O(1) per call). In order to achieve constant time push operation, you can take linear time in pop/top operation.

**2. Approach**

Let us look at the operations: push: As we want push to be constant O(1) operation, hence we will not do anything special, but just enqueue (add) the element in the main queue. Size: Size function is also simple, just return the size of main queue mainQ. Now, let us look at what we should do in pop and top functions, so that we can implement LIFO order (stacks) using the FIFO order in the two queues. Firstly, just print "Stack underflow" and return -1 directly if the main queue (mainQ) size is 0, i.e. there are no elements present. Now, we are sure that there is at least one element present in the main queue to be popped. We need to pop (or peek) the element which is inserted at last (to achieve LIFO order) in the queue. But how to get the last (rear) element of the queue, when we are only allowed to get the front element of the queue (using peek operation)? We will take the help of the auxiliary queue (helperQ) to achieve this task. We will dequeue all the elements (except the last one) from the main queue, one by one, and enqueue them in the auxiliary queue. Now, what we have achieved is that all the elements, except the last element, are in the same order, as they were before, but in the helper queue. We will store the last element in a variable top, and pop this element from the main queue. There will be different lines of code for pop and peek element now: For pop operation, don't need to do anything For peek operation, push this element also in the auxiliary/helper queue. Now, we will do the reverse process, i.e. dequeue elements, one by one, from the helper queue, and enqueue them back to the main queue. Now, return the variable top. Hence, what we achieved by a pair of enqueue-dequeue operations between main and helper queue is, getting the last element from the main queue to pop/peek operation of stack, and making the rest of elements remain in the same order in the end.

**3. Pseudo Code**

1. Push: Add the element in the main queue. 2. Size: Return the size of the main queue. 3. Top: a. If the size of the main queue is 0 (empty), then print "Stack underflow" and return -1. b. Dequeue all the elements until the size of the main queue becomes equal to 1 and Enqueue them to the helper queue. c. Dequeue the last element and store it in a variable top. Enqueue this element also to the helper queue. d. Dequeue all the elements from the helper queue and enqueue them back to the main queue. e. Return the top element. 4. Pop: a. If the size of the main queue is 0 (empty), then print "Stack underflow" and return -1. b. Dequeue all the elements until the size of the main queue becomes equal to 1 and Enqueue them to the helper queue. c. Dequeue the last element and store it in a variable top. d. Dequeue all the elements from the helper queue and enqueue them back to the main queue. e. Return the top element.

**4. Code**

ConsoleJava

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) {

mainQ.add(val);

}

int pop() {

if (size() == 0) {

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

return -1;

} else {

while (mainQ.size() > 1) {

helperQ.add(mainQ.remove());

}

int val = mainQ.remove();

while (helperQ.size() > 0) {

mainQ.add(helperQ.remove());

}

return val;

}

}

int top() {

if (size() == 0) {

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

return -1;

} else {

while (mainQ.size() > 1) {

helperQ.add(mainQ.remove());

}

int val = mainQ.remove();

helperQ.add(val);

while (helperQ.size() > 0) {

mainQ.add(helperQ.remove());

}

return val;

}

}

}

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();

}

}

}

**5. Analysis**

Time & Space Complexity Analysis

Push - O(1): We are just adding the element in the main queue. Size - O(1): We are returning the size of the main queue. Pop (or Top) - O(n): Firstly, we are dequeuing size elements from the main queue adding them in the helper queue, which takes n \* O(1) = O(n) time. Now, we are doing the reverse process (removing n elements from the helper queue and adding them to the main queue), which again takes O(n) time. Hence, total time taken will be O(n + n) = O(n).

## 1. Problem Discussion:

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 for better understanding.

## 2. Approach :

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 Push: Dequeue all elements one by one from the main queue and enqueue them into the helper queue. Add the current element to be pushed into the main queue. Dequeue all elements one by one from the helper queue and enqueue them back into the main queue. Size: Return the size of main queue using mainQ.size() 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(). 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). How about first trying by yourself without reading the code we provide? 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 if you find difficulty in understanding the solution completely.

## 3. Psuedo Code:

Push: Dequeue all elements one by one from the main queue and enqueue them into the helper queue. Add the current element to be pushed into the main queue. Dequeue all elements one by one from the helper queue and enqueue them back into the main queue. Size: Return the size of main queue using mainQ.size() 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(). 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). How about first trying by yourself without reading the code we provide?

## 4. Code

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(); } } }

## 5. Analysis

### 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!