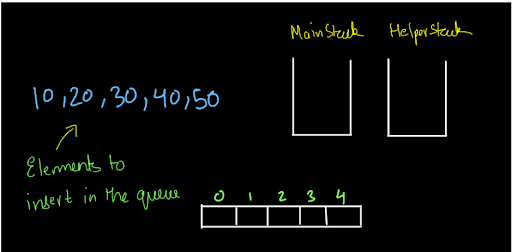
**1. Problem Discussion :**

We are given two stacks, one is the main stack and the other is the helper stack. We have to adapt them and make them behave like a queue. Also, we have to take care that the add method of the queue is efficient i.e. it should be achieved in O(1) time complexity whereas the peek and remove methods can take O(n) time.. The main challenge of this question is to convert the LIFO nature of the stack to the FIFO nature of the queue.

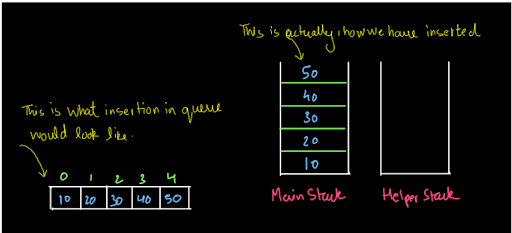
**2. Approach :**

Add :

Let us discuss the add function first. Dear reader, you have already studied that the add method in a queue inserts the value at the end of the queue. Now, let us consider this example given below:

****

Let us say, we want to insert these values into the queue. So, we know that each value will be inserted at the end of the queue. How are we going to insert them? Have a look at fig-2 shown below:

****

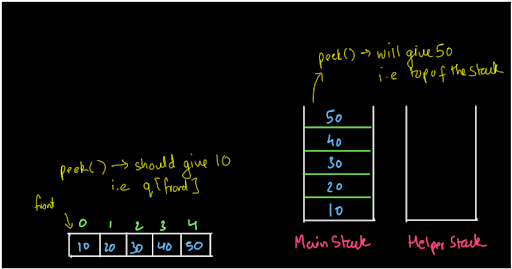
The above diagram shows the analogy of inserting the value in a stack and a queue. The values inserted in a queue would always be inserted at the end of the queue. Similarly, the values inserted in a stack are always at the top of the stack. So, the add function is pretty simple. All we have to do to add a value to a queue is to push the value into the main stack.

Size :

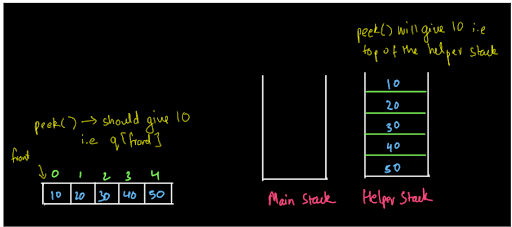
The size function returns the size of the queue at a given time. Since the values inserted in a queue are in reality, being inserted in the main stack given to us, so the size of the queue at any time will be equal to the size of the main stack at that time. So, we just have to return the size of the main stack.

Peek :

So, dear reader, did you get the above two methods? If you have any doubts about the above methods, you may refer to the solution video to clear all your doubts regarding the add and size methods. The making of add method efficient will lead to an increase in the time of the peek and remove methods. Why? Think about it yourself!!!

****

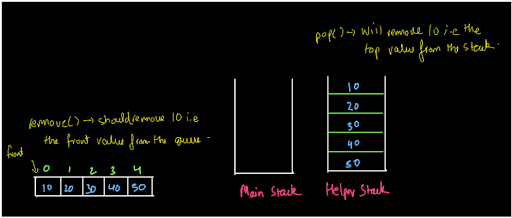
The problem is pretty obvious. If we peek into the queue, we should get the element at the front of the queue i.e. we should get 10. But, if we apply the peek method directly to the main stack, we will get the value 50 as the peek method of the stack will return the element at the top of the stack. So, what should we do? Let us bring the other stack (helper stack) into the picture now. Let's pop all the elements one by one from the main stack and push them into the helper stack. So, first of all, 50 will be popped out from the main stack and will be inserted into the helper stack. So, the element which was at the top of the main stack is now at the bottom of the helper stack and the element which was at the bottom, now is at the top of the helper stack. Have a look at the diagram shown below (fig-4):

****

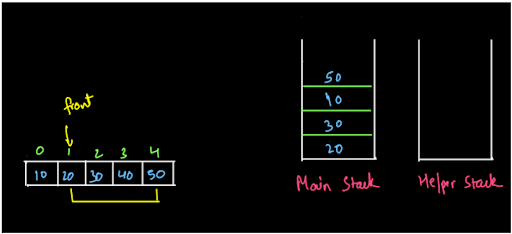
So, if we peek into the helper stack, the element at the top is actually the element at the front of the queue. So, we will now peek from the helper stack and it will be analogous to the peek method of the queue. After this, we will copy back the elements from the helper stack to the main stack and the situation will again be like fig-3. So, the peek method is also complete. Now, let us move to the last method i.e. remove.

Remove :

The queue always removes the value from its front. But we know that the top of the main stack gives us the rear and not the front. So, how will we remove the front element? We will apply the same procedure as we did for the peek method. Let us copy all the elements from the main stack to the helper stack (see fig-5)

****

Now, if we pop the element from the helper stack then 10 will be removed and this is what we wanted. So, we will pop from the helper stack and copy the values back to the main stack. The corresponding queue and situation of the main stack after popping is shown below:

****

So, this is how we will remove the element from the queue adapted from two stacks.

Pseudo Code :

Add: The add method is very easy to implement. To add the values in the queue we will push the values into the main stack. Peek: Keep on popping out the values from the main stack and inserting them into the helper stack. When the main stack is empty, return the value at the top of the helper stack and copy the values back to the main stack. Size: It is also very simple. Just return the size of the main stack. Remove: Do the same as we did for the peek method except, before copying the values back to the main stack, pop once from the helper stack and then copy.

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

**3. Code :**

ConsoleJava

import java.io.\*;

import java.util.\*;

public class Main {

public static class StackToQueueAdapter {

Stack< Integer> mainS;

Stack< Integer> helperS;

public StackToQueueAdapter() {

mainS = new Stack< >();

helperS = new Stack< >();

}

int size() {

return mainS.size();

}

void add(int val) {

mainS.push(val);

}

int remove() {

if (size() == 0) {

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

return -1;

} else {

while (mainS.size() > 1) {

helperS.push(mainS.pop());

}

int val = mainS.pop();

while (helperS.size() > 0) {

mainS.push(helperS.pop());

}

return val;

}

}

int peek() {

if (size() == 0) {

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

return -1;

} else {

while (mainS.size() > 1) {

helperS.push(mainS.pop());

}

int val = mainS.pop();

helperS.push(val);

while (helperS.size() > 0) {

mainS.push(helperS.pop());

}

return val;

}

}

}

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

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

StackToQueueAdapter qu = new StackToQueueAdapter();

String str = br.readLine();

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

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

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

qu.add(val);

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

int val = qu.remove();

if (val != -1) {

System.out.println(val);

}

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

int val = qu.peek();

if (val != -1) {

System.out.println(val);

}

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

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

}

str = br.readLine();

}

}

}

**4. Analysis :**

Time Complexity :

Add: The time complexity is O(1) because we have used the stack push method. Size: The time complexity is O(1) as we have just returned the size of the stack. Peek: The time complexity of the peek method is O(n) as we pooped the entire stack of n elements and then pushed them back again. So, there are two traversals. So, the time complexity will be n+n=2n i.e. O(n). Remove: The time complexity of this method is O(n) as we pooped the entire stack of n elements and then pushed them back again. So, there are two traversals. So, the time complexity will be n+n=2n i.e. O(n).

Space Complexity :

Well the space complexity can be considered as O(n) as we are using two stacks for implementing the queue. But, these two stacks are given to us in our question, and apart from these stacks, we have not used any extra data structure or memory to implement the queue. Since the data structure that is given in the question (2 stacks in this case) is not considered in the space complexity, the space complexity will be O(1). We hope that you got the solution completely and also understood that making add efficient will increase the time for peek and remove methods. If you still have any doubts, you may refer to the complete solution video to clear all of them.

**5. Suggestions :**

Here are some suggestions from our side that you do not want to miss:

1• The next question is about doing the same thing but making the remove method efficient. We recommend you try to solve it on your own first. Take hints from this question and you will very easily solve the next one. 2• The stacks and queues are simple yet tricky data structures because of their LIFO and FIFO properties respectively. Later in this course and in LEVEL 2 when you will solve problems, it will be very crucial to understand what to use. So, these questions are clearing your basics to get you ready for such problems. We suggest you revise the entire section in one go after the completion of the next two questions which will conclude the section. Until then, Happy Coding!!!

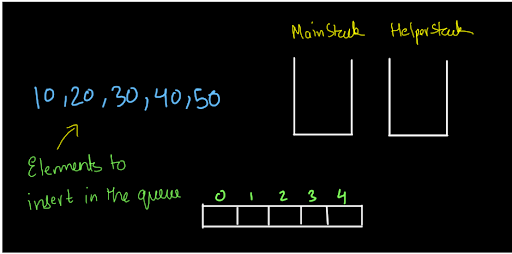
**1. Problem Discussion :-**

We are given two stacks, one is the main stack and the other is the helper stack. We have to adapt them and make them behave like a queue. Also, we have to take care that the remove method of the queue is efficient i.e. it should be achieved in O(1) time complexity whereas the add method can take O(n) time. If you have solved the previous problem then this is just a piece of cake for you.

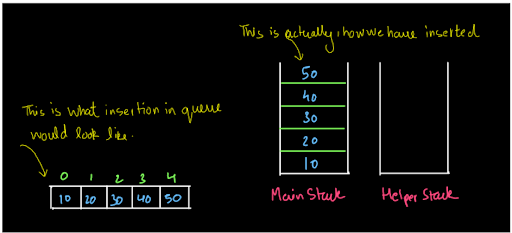
You may refer to the question video to understand the problem completely. The main challenge of this question is to convert the LIFO nature of the stack to the FIFO nature of the queue.

**2. Approach :-**

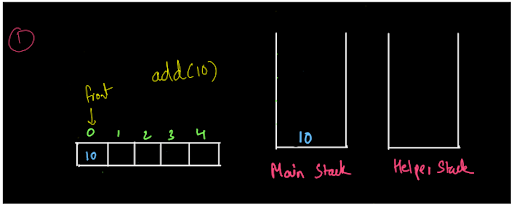
Add: Let us consider the following example.

****

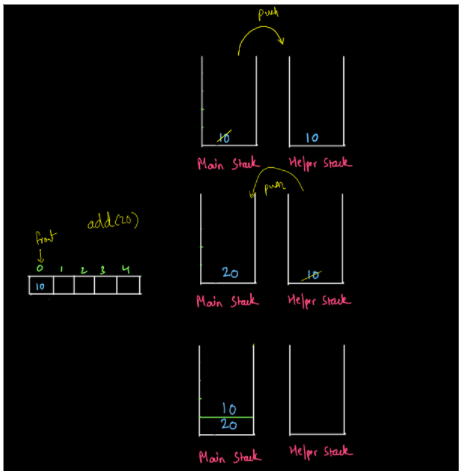
We have to add these elements to the queue. If we simply add these elements to the main stack then it will be analogous to add them in a queue as adding the elements to the end of the queue is analogous to adding the elements at the top of the stack.

****

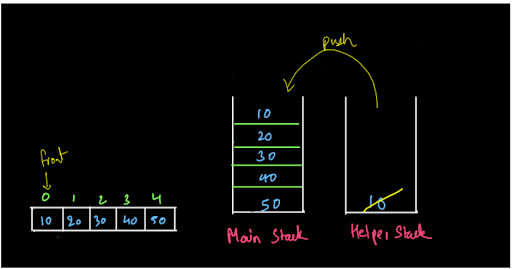
This is what we did in the previous question when we had to make the add method efficient without taking care of the remove method. Here, the situation is exactly the opposite. If you notice, when we try to remove an element from the queue, we will not be able to remove the element in O(1) time (You know this from the previous question). So, it is clear that we will not insert the elements in this way. So, the question is, how are we going to insert the elements?

****

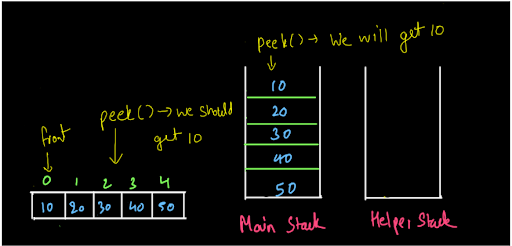
We are trying to insert the same elements as shown above step by step. So, we have inserted the first element 10 into the main stack. Our aim is that when we pop from the main stack, we should get the element at the front of the queue. Why? Well obviously, to make the remove function efficient. If we see in this case the front of the queue is at the top of the main stack. So, there is no problem.

****

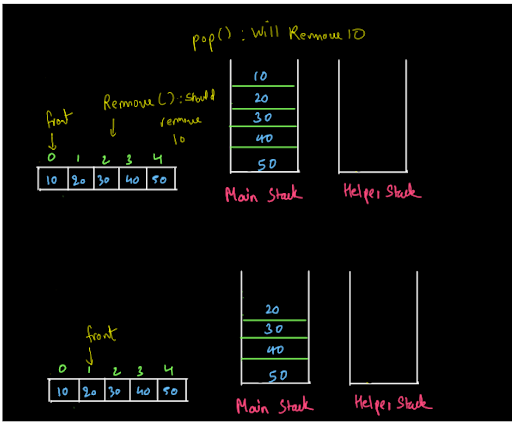
Now we try to add 20 into the queue. So, if we push 20 into the main stack, then the top of the main stack will contain the rear element of the queue rather than the front. So, we pop out all the contents of the main stack and push it into the helper stack. Then, we push the value i.e. 20 into the main stack and pop all the values from the helper stack, and push them into the main stack. We find that the main stack now has the front element of the queue at its top and whenever we try to remove it from the queue, we will be able to remove the front element easily by popping the value from the main stack. We will keep on doing this and after all the elements are added, our stacks will look like this:

****

So, finally, we conclude that to add an element to the queue we will pop out all the elements present in the main stack and push them into the helper stack and then push the given element into the main stack and again pop out all the values from the helper stack and push them into the main stack. You may refer to the solution video (0:29-2:16) if you have any doubts regarding the add method. Size: The size of the queue will be equal to the size of the main stack at any particular time. So, it is going to be very easy. We just have to return the size of the main stack. Peek: As we have already discussed, we have added the elements in such a way that the element at the front of the queue will be at the top of the stack. So, we just have to apply the peek method to the main stack which will return the topmost element of the stack i.e. q[front]

****

Remove: Again, we did all the hard work in the add function to make the remove method efficient. So, we know that we just have to pop the element from the main stack to remove an element from the queue (removing the element at the front in the queue).

****

**3. CODE -:**

ConsoleCpp

#include<iostream>

#include<bits/stdc++.h>

using namespace std;

class StackToQueueAdapter {

public:

stack <int> mainS;

stack <int> helperS;

int size() {

return mainS.size();

}

void add(int val) {

while (mainS.size() > 0) {

helperS.push(mainS.top());

mainS.pop();

}

mainS.push(val);

while (helperS.size() > 0) {

mainS.push(helperS.top());

helperS.pop();

}

}

int Remove() {

if (mainS.size() == 0) {

cout << "Queue underflow" << endl;

return -1;

}

int tr = mainS.top();

mainS.pop();

return tr;

}

int peek() {

if (mainS.size() == 0) {

cout << "Queue underflow" << endl;

return -1;

}

return mainS.top();

}

};

int main() {

string str;

StackToQueueAdapter qu;

while (true) {

getline(cin, str);

if (str[0] == 'q') {

break;

}

else if (str[0] == 'a') {

string ss = str.substr(4, 2);

int n = stoi(ss);

qu.add(n);

}

else if (str[0] == 's') {

cout << qu.size() << endl;

}

else if (str[0] == 'r') {

int val = qu.Remove();

if (val != -1) {

cout << val << endl;

}

} else if (str[0] == 'p') {

int val = qu.peek();

if (val != -1) {

cout << val << endl;

}

}

}

return 0;

}

**4. Pseudo Code -:**

Add: We will not implement the add method directly. Rather, we will first pop all the elements from the main stack and push them into the helper stack. Then, we will add the new element to the main stack and pop all the elements from the helper stack, and push them to the main stack.

Peek: We will use the peek method of the stack and it will give us the top value of the main stack. This will be the same as the front value of the queue.

Size: It is also very simple. Just return the size of the main stack.

Remove: Simply pop a value from the main stack. This will remove the value from the front of the queue.

**5. Analysis -:**

Time Complexity : O(1)

Add: The time complexity is O(n) as we are popping all the values from the main stack and pushing them into the helper stack and doing vice-versa. So, the time complexity becomes n+n=2n=O(n). Remove: The time complexity for this method is O(1) as we just popped a value from the main stack. Size: The time complexity is O(1) as we just return the size of the main stack. Peek: The time complexity is O(1) as we just use the peek method of the main stack.

Space Complexity : O(1)

Well the space complexity can be considered as O(n) as we are using two stacks for implementing the queue. But, these two stacks are given to us in our question, and apart from these stacks, we have not used any extra data structure or memory to implement the queue. Since the data structure that is given in the question (2 stacks in this case) is not considered in the space complexity, the space complexity will be O(1).

That was easy. Wasn't it? Our desire to make you learn will remain unsatisfactory if you still have doubts. We strongly recommend you to watch our video lecture on Stack to Queue - Remove Efficient for clearing any type of doubts. Suggestions and feedback are always welcomed. You can contact us via our website. All the best for a bright future! Happy Coding!