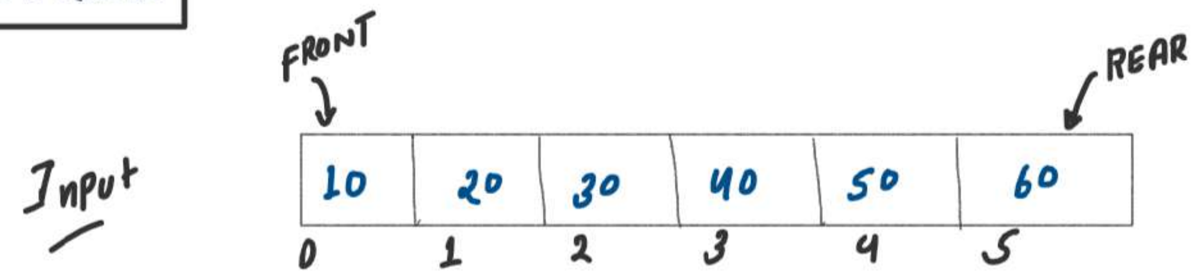
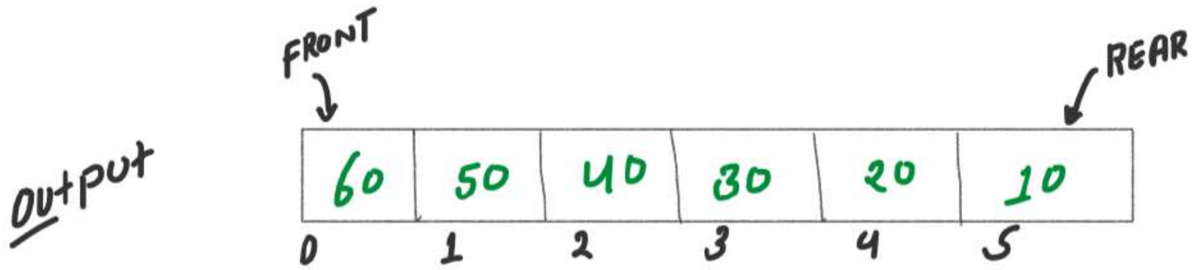
22/11/2023

QUEUE CLASS - 2

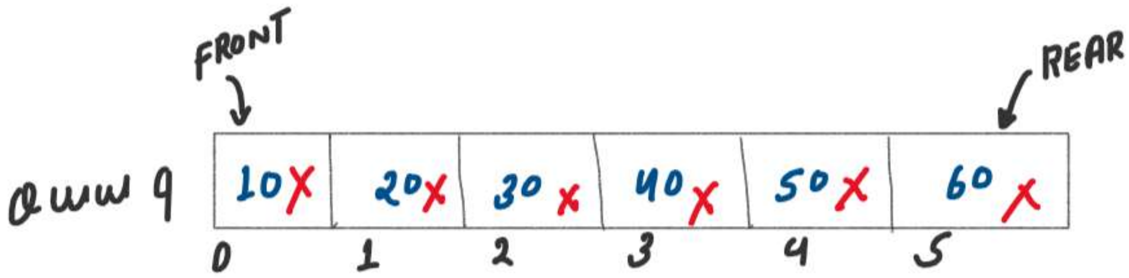
1. Reverse a Queue

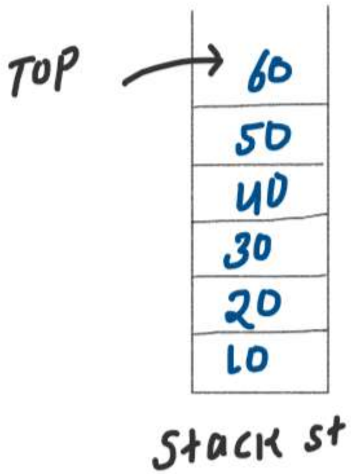


APPROACH 1: USING STACK
APPROACH 2: USING RECURSION



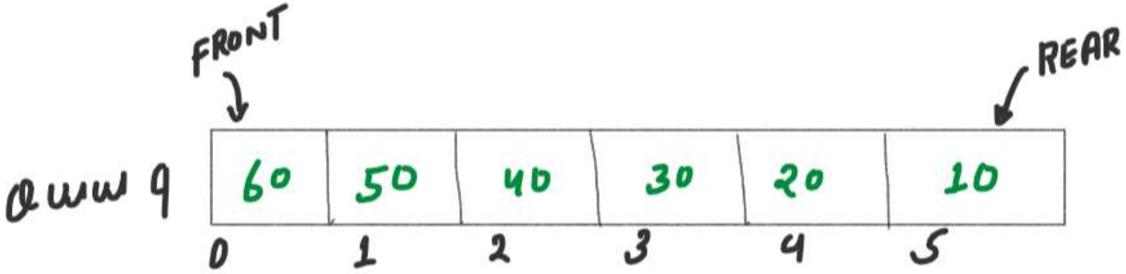
APPROACH 1: USING STACK





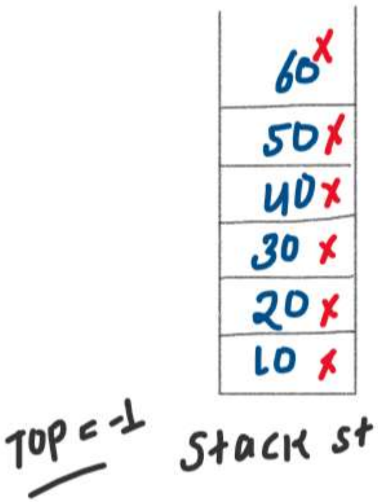
Step1 One by one queue se element 10 and stack me insert kardo

```
while ( !q.empty() ) {
        int frontElement = q.front();
        q.pop();
        s.push( frontElement );
}
```



step2 One by one stack se element 10 and queue me insert karo

```
while ( !st.Empty() ) {
        int TOPElement = st.top();
        s.pop();
        q.push( TOPElement);
}
```



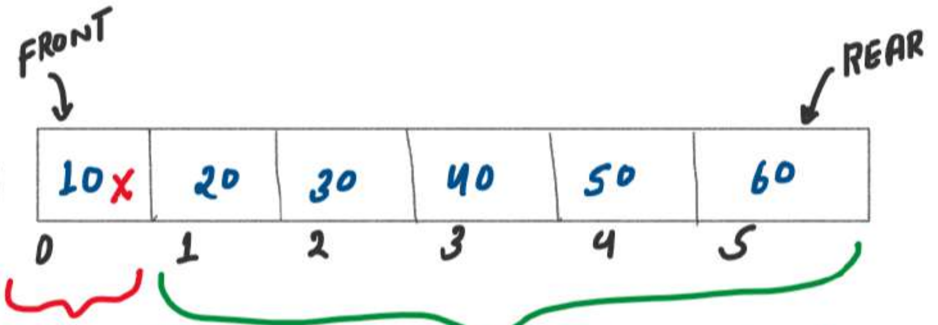
```cpp
// 1. Reverse a queue
// APPROACH 01: USING STACK

void reverseQueue(queue<int> &q){
    stack<int> st;

    // Step 1: one by one queue se element lelo and stack me insert kra do
    while(!q.empty()){
        int frontElement = q.front();
        q.pop();

        st.push(frontElement);
    }

    // Step 2: one by one stack se element lelo and queue me insert kra do
    while(!st.empty()){
        int topElement = st.top();
        st.pop();

        q.push(topElement);
    }
}
```

Time Complexity: *O(N),*
Where N is numbers of elements in queue

Space Complexity: *O(N),*
Where stack stores N elements from the queue.

APPROACH 2: USING RECURSION

Queue q



HUM SOLVE KAR LENGE

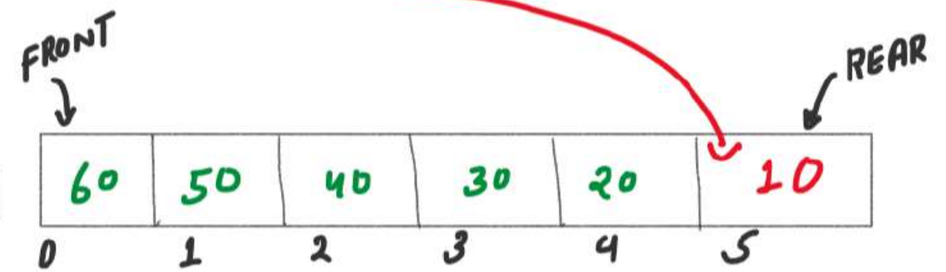
RECURSION SOLVE KAR LEGA

```
int temp = Q.front();
Q.pop();
f(Q);
Q.push(temp);
```

Base Case

```
if(q.empty())
    return
```

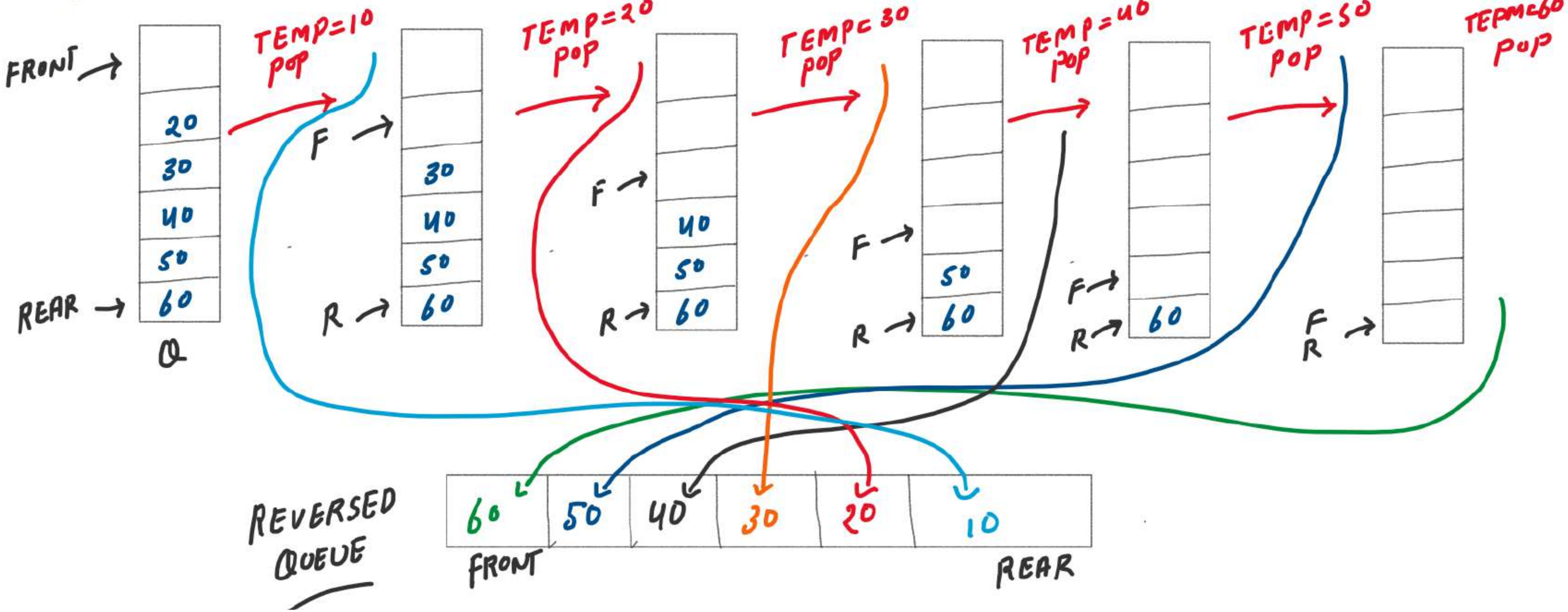
q



```cpp
// 1. Reverse a queue
// APPROACH 02: USING RECURSION

void reverseQueueRE(queue<int> &q){
    // Base case
    if(q.empty()) return;

    // Ek step hum solve kar lenge
    int temp = q.front();
    q.pop();

    // Recursion solve kar lega
    reverseQueueRE(q);
    q.push(temp);
}
```

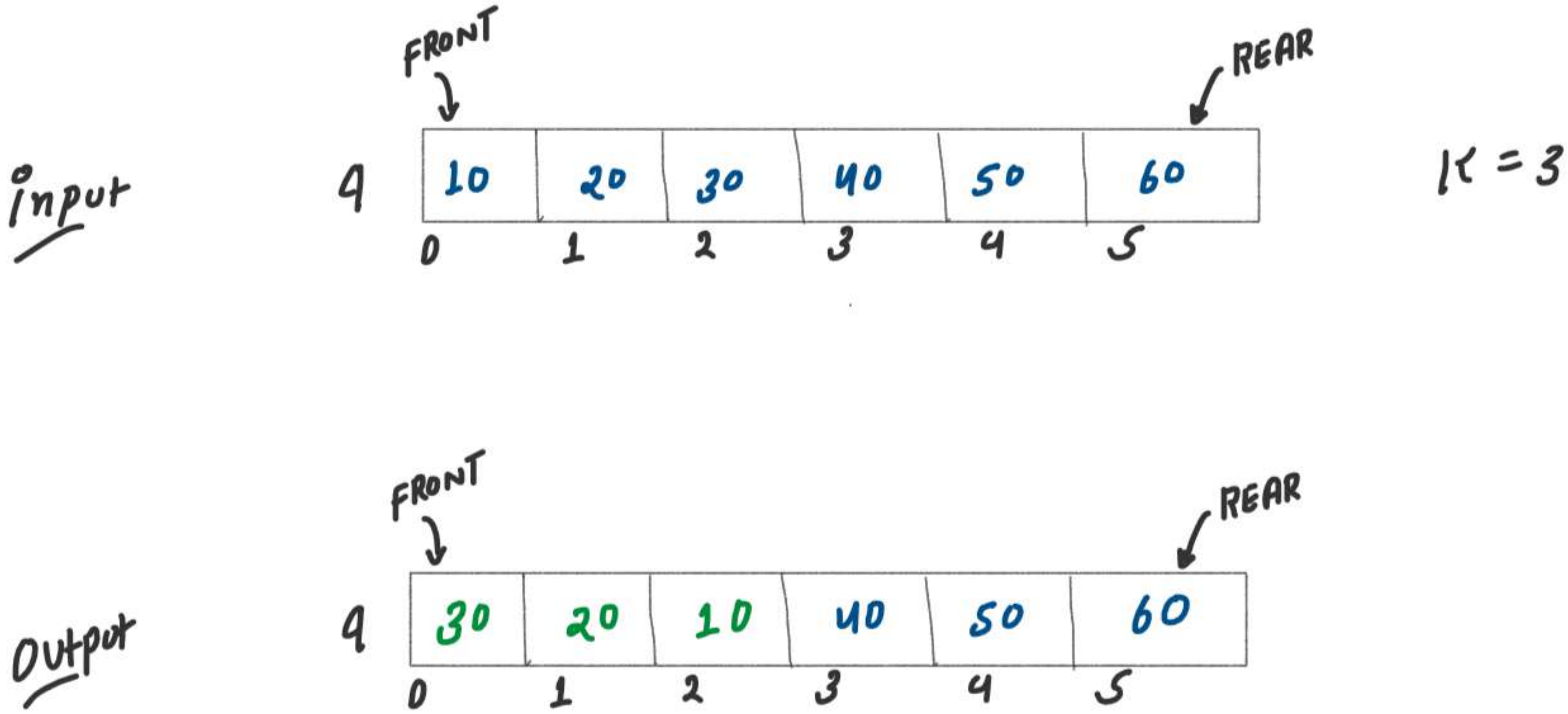
Time Complexity: *O(N)*
Space Complexity: *O(N)*
Where N is numbers of elements in queue

DRY RUN

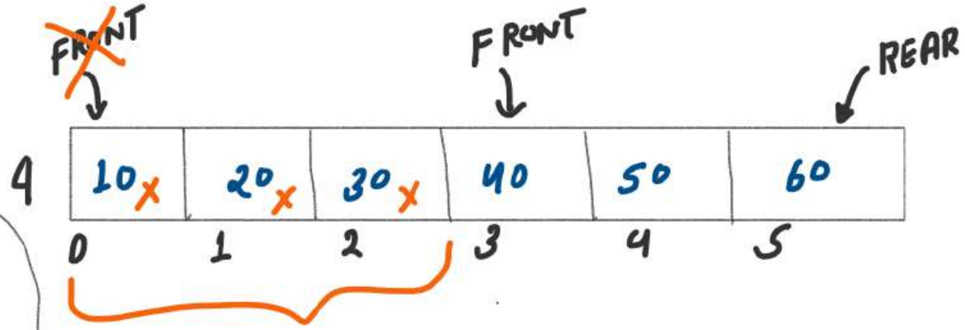


FRONT
REAR
Q
20
30
40
50
60
TEMP=10
pop
F
R
TEMP=20
pop
TEMP=30
pop
TEMP=40
pop
TEMP=50
pop
TEMP=60
pop
REVERSED QUEUE
60
50
40
30
20
10
FRONT
REAR

2. Reverse K elements in a queue



Approach



$K = 3$

$N = \text{Queue size} = 6$

Step1

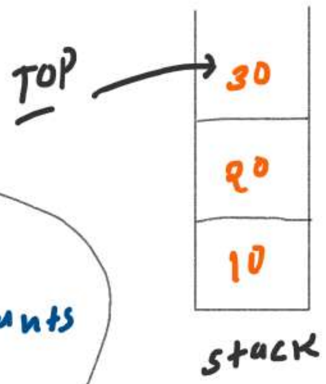
push K Elements
from Queue to stack

Step2

push K Elements
from stack to Queue

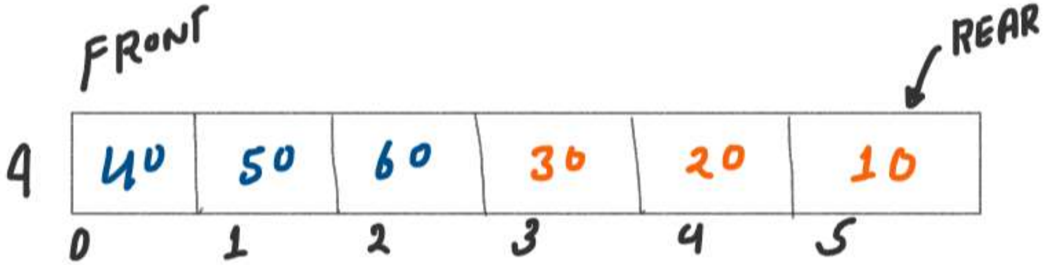
Step3

pop and push first N-K Elements
from Queue to Queue



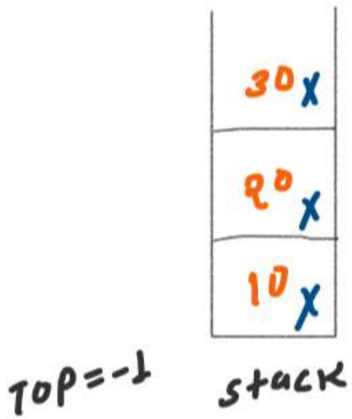
Step:1

```
for( int i=0 ; i<K ; i++){
    int FrontE = q.front();
    q.pop();
    st.push(frontE);
}
```



K = 3

N = Queue size = 6



step:2

```
for( int i=K ; i<N ; i++){
    int TOPE = st.top();
    st.pop();
    q.push(TOPE);
}
```

N = 6

FRONT REAR

q					
40	50	60	30	20	10
0	1	2	3	4	5

N-K = 3

K = 3

N = Queue size = 6

Step3

```
for(int i=0; i<(N-K); i++)
{
    int FrontE = q.front();
    q.pop();
    q.push(FrontE);
}
```

FRONT REAR

q					
30	20	10	40	50	60
0	1	2	3	4	5

Output

```cpp
// 2. Reverse 'k' element in a queue ⭐
// APPROACH: USING STACK

void reverseKEQueue(queue<int> &q, int K){
    stack<int> st;
    int N = q.size();

    // Step 1: push K element from queue to stack
    for(int i=0; i<K; i++){
        int frontElement = q.front();
        q.pop();
        st.push(frontElement);
    }

    // Step 2: push K element from stack to queue
    for(int i=K; i<N; i++){
        int topElement = st.top();
        st.pop();
        q.push(topElement);
    }

    // Step 3: pop and push first (N-K) elements from queue to queue
    for(int i=0; i<(N-K); i++){
        int frontElement = q.front();
        q.pop();
        q.push(frontElement);
    }
}
```

APPROACH 01: USING STACK

Time Complexity: *O(N),*
Where N is numbers of elements in queue

Space Complexity: *O(N),*
Where N is numbers of elements in queue

📂 3. Interleave first and second half of a queue

Input

	FRONT							REAR
q	10	20	30	40	50	60	70	80

Output

	FRONT							REAR
q	10	50	20	60	30	70	40	80

Approach

First → q

FRONT							REAR
10 x	20 x	30 x	40 x	50	60	70	80

$\boxed{size = 8}$

Step 1 Break queue into two half

Second

FRONT			REAR
10	20	30	40

```
for( int i=0; i<(size/2); i++){
      int frontE = First.front();
      First.pop();
      second.push(frontE);
}
```

Step 2 MERGE Both Half

```
while(!second.empty()){
    int sFront = second.front();
    second.pop();
    first.push(sFront);
    int fFront = first.front();
    first.pop();
    first.push(fFront);
}
```

first (~~FRONT~~ ... ~~REAR~~)

50 X	60 X	70 X	80 X

second (~~FRONT~~ ... ~~REAR~~)

10 X	20 X	30 X	40 X

First as MERGE (FRONT ... REAR)

10	50	20	60	30	70	40	80

Output

```cpp
// 3. Interleave first and second half of a queue
// APPROACH 01: ITERATIVE

void interLeaveQueue(queue<int> &first){
    queue<int> second;
    int size = first.size();

    // Step 1: break queue into half
    for(int i=0; i<(size/2); i++){
        int fFront = first.front();
        first.pop();
        second.push(fFront);
    }

    // Step 2: merge both half
    while(!second.empty()){
        int sFront = second.front();
        second.pop();
        first.push(sFront);

        int fFront = first.front();
        first.pop();
        first.push(fFront);
    }
}
```

APPROACH 01: ITERATIVE

Time Complexity: *O(N),*
Where N is numbers of elements in queue

Space Complexity: *O(N),*
Where N is numbers of elements in queue

📂 4. First negative integer in every window of K elements

Most Important (window sliding pattern)

Input

arr

2	-5	4	-1	-2	0	5
0	1	2	3	4	5	6

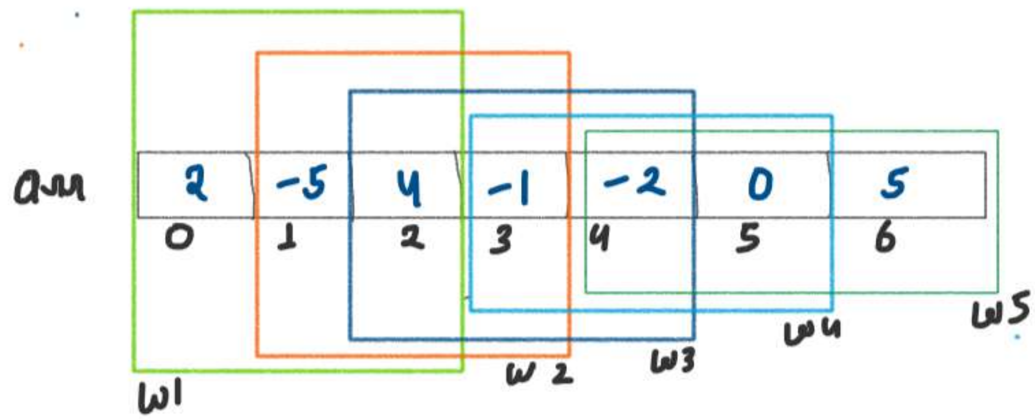
K = 3

output

-5	-5	-1	-1	-2

Explanation

$\boxed{K=3}$



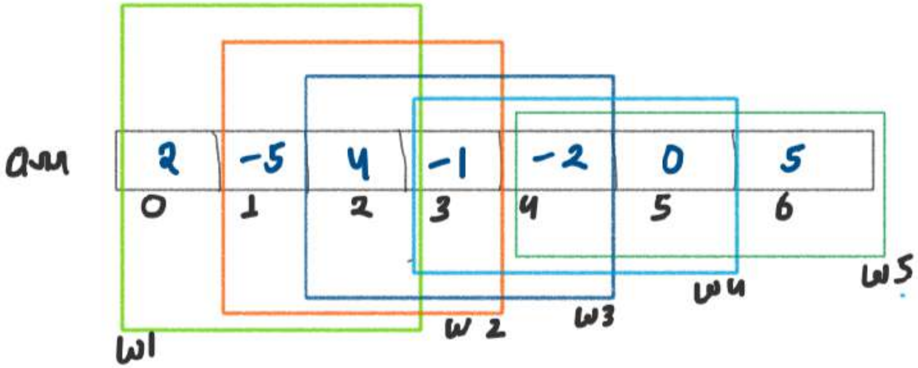
first -ve

w1	2	-5	4	= -5
w2	-5	4	-1	= -5
w3	4	-1	-2	= -1
w4	-1	-2	0	= -1
w5	-2	0	5	= -2

Condition

window me Agar negative nahi milta hai To 0 print kar dingy.

APPROACH



K=3
N=size = 7

Step1

Process first K Element of first window

W1

2	-5	4
0	1	2

First negative number

Double QUEUE

1

FRONT

store the index of first -ve Element.

why store Index?

why use Double queue?

Step2 Process first K Element of remaining window

$\overbrace{\qquad\qquad\qquad}^{w2}$

~~2~~	-5	4	-1
0	1	2	3

REMOVE OLD ELEMENT (index 0)

INSERT NEW ELEMENT (index 3)

Double QUEUE

1	3	
FRONT	REAR	

Why stored Index?

Ans ⇒ to check the out of Range of Element ki hume next Remaining windows ka kab Index Remove karna hai and kab index Insert karna hai in the Double queue.

K=3
Index = 3

REMOVE Q.front() when

$Index - Q.front() >= K$

$3 - 1 >= 3$

$2 >= 3$ False

Insert Q.front() when

$(Index - Q.front < K)$ $\Big\{$ (arr[index]<0)

$2 < 3$ TRUE REAR $\boxed{3}$

DRY RUN

arr	2	-5	4	-1	-2	0	5
	0	1	2	3	4	5	6

W1 W2 W3 W4 W5

K=3
N=size=7

Insertion Range [0,3)

0,1,2 < 3

(Index − Q.front() < k)
(arr[index] < 0)

W1	2	-5	4
	0	1	2

FRONT
Q | 1 |

W2	-5	4	-1
	1	2	3

Index
Q | 1 | 3 | → 3 − 1 ⇒ 2 < 3 ✓ { -1 < 0 ✓

W3	4	-1	-2
	2	3	4

Index
Q | 3 | 4 | → 4 − 1 ⇒ 3 < 3 X { -2 < 0 ✓

W4	-1	-2	0
	3	4	5

Index
Q | 3 | 4 | → 5 − 3 ⇒ 2 < 3 ✓ { 0 < 0 X

W5	-2	0	5
	4	5	6

Index
Q | 4 | → 6 − 3 ⇒ 3 < 3 X { 5 < 0 X

why use Double Queue?
↳ To insert and remove from both End.

```cpp
// 4. First negative integer in every window of 'k' ⭐
#include<iostream>
#include<deque>
using namespace std;

void printFirstNegative(int *arr, int size, int k){
    deque<int> dq;
    // Step 1: Process the first k elements in first window
    for(int i = 0; i < k; i++){
        int element = arr[i];
        if(element < 0){
            dq.push_back(i);
        }
    }

    // Step 2: Process the first k elements in next remaining windows
    for(index = k; index < size; index++){
        // Aage badne se pahle --> Print first negative element of old windows
        if(dq.empty()){
            cout<<" 0"<<" ";
        }
        else{
            cout<< arr[dq.front()] << " ";
        }
        // Remove Old Index From Queue When (index - dq.front() >= k)
        if(index - dq.front() >= k){
            dq.pop_front();
        }
        // Insert New Index From Queue When (arr[index] < 0)
        if(arr[index] < 0){
            dq.push_back(index);
        }
    }

    // Print first negative element of last window
    if(dq.empty()){
        cout<<" 0"<<" ";
    }
    else{
        cout<< arr[dq.front()] << " ";
    }
}
```

APPROACH 1: USING QUEUE (Window Sliding Pattern)

Time Complexity: *O(N), where N is size of array*
Space Complexity: *O(K), where K is the size of the window*