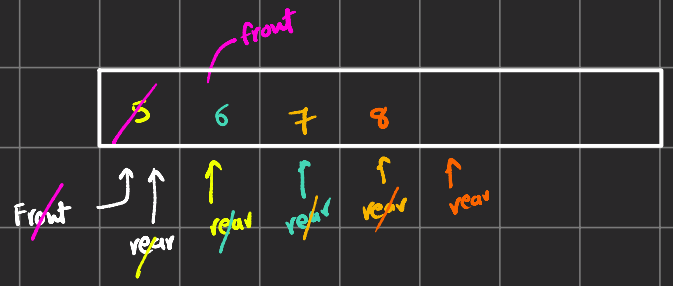
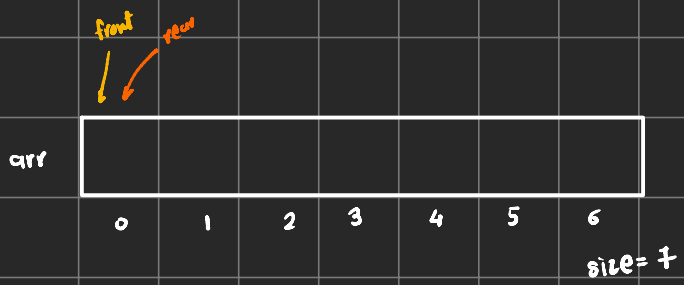
✱ Queue → data structure FIFO - First In First Out



insertion from → rear
pop from → front

✱ Implement



Push

if (Queue is full) → Q is full
else ↳ insert data

if (rear == size) → Q is full
else → arr[rear] = data
rear++

Pop

if (Queue is empty) → Q is empty
else ↳ pop data

if (front == rear) → Q is empty
else ↳
arr[front] = -1
front ++
if (front == rear) → Queue empty ho gyi
↳ front = 0
rear = 0

getFront

if (queue is empty) → empty
else return front

isEmpty

if (front == rear) → empty
else → not empty

* circular queue

front = -1
rear = -1

first Element insertion } handle
single Element removal } seperately

Push

if (Q is full) → Q full

else if (first Ele insert) → f=r=0 then insert

else if ↳ establish circular nature →

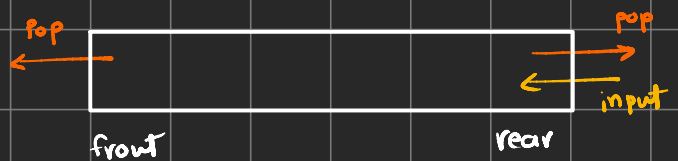
if (rear = n-1 && front != 0) → rear = 0 and insert

else ↳ default nature → rear++
arr[rear] = Ele

```
void pop(){

    // empty check
    if(front == -1){
        cout << "q is empty cannot pop" << endl;
    }

    // single ele
    else if(front==rear){
        arr[front] = -1;
        front = -1;
        rear = -1;
    }

    // circular nature
    else if(front == size-1){
        front = 0;
    }

    // normal flow
    else{
        front++;
    }

}
```

```
class CricularQueue{

    public:
        int size;
        int* arr;
        int front;
        int rear;

        CricularQueue(int size){
            this->size = size;
            arr = new int[size];
            front = -1;
            rear = -1;
        }

        void push(int data){

            // queue is full
            if( (front == 0 && rear == size-1) || (rear == (front-1)%(size-1) ) ){
                cout << "queue is full cannot insert" << endl;
            }

            // single ele case -> first ele insert
            else if( front==-1 && rear==-1 ){
                front = 0;
                rear = 0;
                arr[rear] = data;
            }

            // circular nature
            else if( rear == size-1 && front != 0 ){
                rear = 0;
                arr[rear] = data;
            }

            // normal queue
            else{
                rear++;
                arr[rear] = data;
            }
        }
```

① input restricted queue



rear → input (push_back)

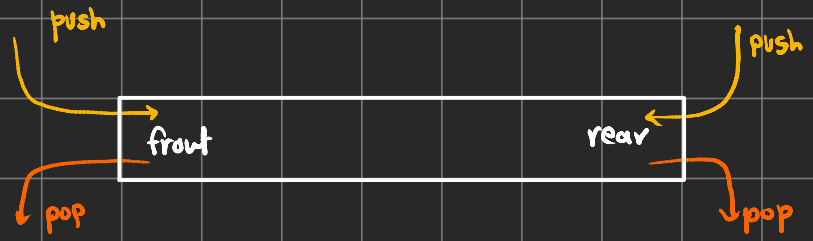
pop → front (pop_front)
pop → rear (pop_back)

② Output Restricted Queue

pop → front

push → front
push → rear

* Doubly ended Queue (deque - dek)



dequeue → pop
enqueue → push

```
class Deque{

    public:
        int* arr;
        int size;
        int front;
        int rear;

        Deque(int size){
            this->size = size;
            arr = new int[size];
            rear = -1;
            front = -1;
        }

        void pushRear(int data){
            // queue is full
            if( (front == 0 && rear == size-1) || (rear == (front-1)%(size-1) ) ){
                cout << "queue is full cannot insert" << endl;
            }

            // single ele case -> first ele insert
            else if( front==-1 && rear==-1 ){
                front = 0;
                rear = 0;
                arr[rear] = data;
            }

            // circular nature
            else if( rear == size-1 && front != 0 ){
                rear = 0;
                arr[rear] = data;
            }

            // normal queue
            else{
                rear++;
                arr[rear] = data;
            }
        }
```

```cpp
void pushFront(int data){
    // queue is full
    if( (front == 0 && rear == size-1) || (rear == (front-1)%(size-1) ) ){
        cout << "queue is full cannot insert" << endl;
    }

    // single ele case -> first ele insert
    else if( front==-1 && rear==-1 ){
        front = 0;
        rear = 0;
        arr[front] = data;
    }

    // circular nature
    else if( rear != size-1 && front == 0 ){
        front = size-1;
        arr[front] = data;
    }

    // normal queue
    else{
        front--;
        arr[front] = data;
    }
}
```

```cpp
void popFront(){
    // empty check
    if(front == -1){
        cout << "q is empty cannot pop" << endl;
    }

    // single ele
    else if(front==rear){
        arr[front] = -1;
        front = -1;
        rear = -1;
    }

    // circular nature
    else if(front == size-1){
        front = 0;
    }

    // normal flow
    else{
        front++;
    }
}

void popRear(){
     // empty check
    if(front == -1){
        cout << "q is empty cannot pop" << endl;
    }

    // single ele
    else if(front==rear){
        arr[front] = -1;
        front = -1;
        rear = -1;
    }

    // circular nature
    else if(rear == 0){
        rear = size - 1;
    }

    // normal flow
    else{
        rear--;
    }
}
```

Queue Reverse

ip: | 3 | 6 | 9 | 2 | 8 |

op: | 8 | 2 | 9 | 6 | 3 |

put in stack

8
2
9
6
3

8	2	9	6	3

Tc. $O(2n) = O(n)$

Sc. $O(n)$ (stack)

Stack se kar sakte hun to recursion se bhi kar sakte hun

3	6	9	2	8

1st step — Rec will take care

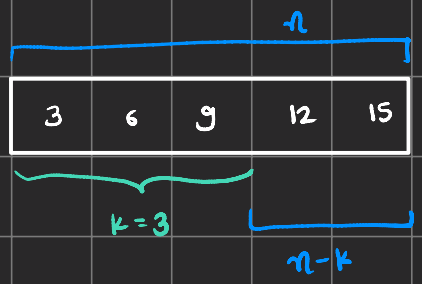
→ 3 ko nikal lo
Recursion de do
3 ko piche daal do

```
① { int temp = q.front();
    q.pop()
② { Recursion(q)
③ { q.push(temp)
```

~~3~~	6	9	2	8

6	9	2	8	3

★ Reverse first K elements



k = 3

9	6	3	12	15

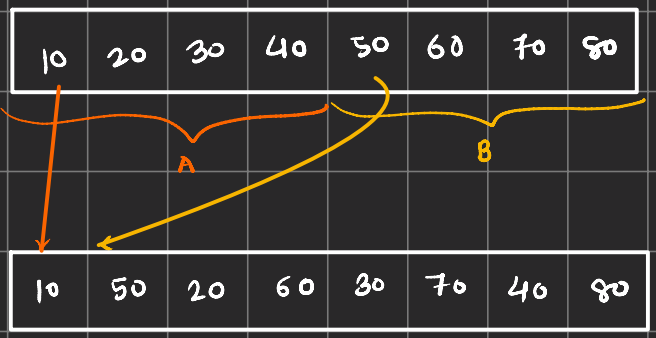
→ k element → stack

→ stack → queue

→ (n-k) ele ⇝ pop then push

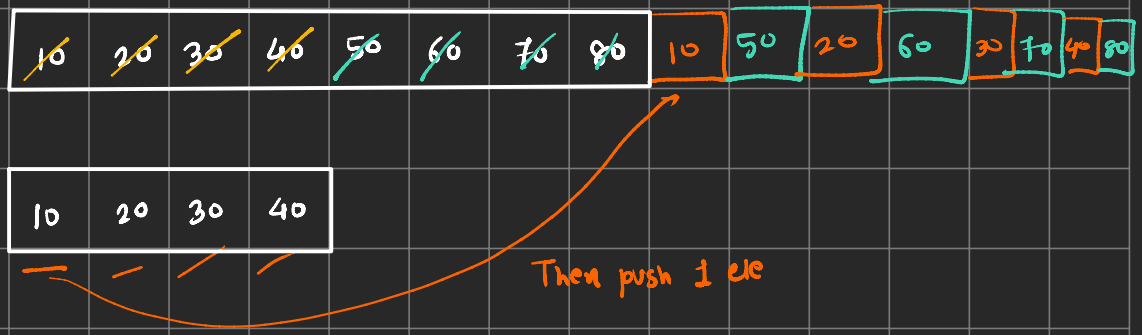
★ Interleave first and second half of queue

→ Transfer n/2 eles from queue to another queue



This is called interleave

10	50	20	60	30	70	40	80



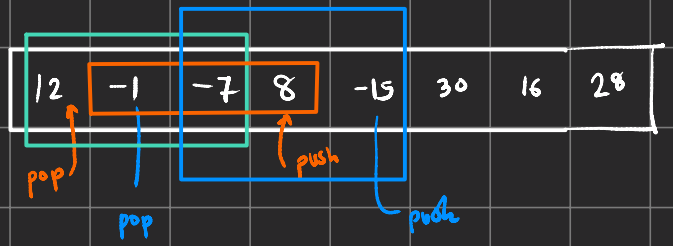
✱ Sliding Window

✱ First -ve integer in every window of size K

K = 3

Array.

12	-1	-7	8	-15	30	16	28



{ -1, -1, -7, -15, -15, 0 }

when you do not get any -ve then

K = 2

-8	2	3	-6	10

first window

deque<int> dq

① First K sized window process (First window) ⇝ -ve index inside push dq.

dq.

0	

answer = dq.front()

② Processing remaining window

{ -8, 2, 3, -6, 10 }

K n-1

→ Removal

→ for (k → <n) → Removal if (dq.front() - i >= k) → dq.pop_front()

→ Addition if (arr[i] < 0) → dq.push_back(i)

```cpp
void solve(int arr[], int size, int k)
{
    vector<int> ans;
    // Create a deque
    deque<int> q;
    // Process first window of size = k
    for (int i = 0; i < k; i++)
    {
        // If negative number found, then simply insert index in queue
        if (arr[i] < 0)
        {
            q.push_back(i);
        }
    }
    // Process the remaining windows
    for (int i = k; i < size; i++)
    {
        // First store answer of previous window
        if (q.empty())
        {
            ans.push_back(0);
        }
        else
        {
            int temp = arr[q.front()];
            ans.push_back(temp);
        }
        // Out of window element to be removed
        while (!q.empty() && (i - q.front()) >= k)
        {
            q.pop_front();
        }
        // Check current element whether negative or not. If negative then push index in queue
        if (arr[i] < 0)
        {
            q.push_back(i);
        }
    }
    // For last window (as last window was not processed)
    if (q.empty())
    {
        ans.push_back(0);
    }
    else
    {
        int temp = arr[q.front()];
        ans.push_back(temp);
    }
    // Printing ans vector
    for(int i = 0;i<ans.size();i++){
        cout<<ans[i]<<" ";
    }
}
```

→ Answer → dq.size() != 0 → ans → dq.front()

→ size = 0 → 0

```cpp
deque<long long int> dq;
vector<long long> ans;
int negative = -1;

//process first window
for(int i=0; i<K; i++) {
    if(A[i] < 0) {
        dq.push_back(i);
    }
}

//push ans for FIRST window
if(dq.size() > 0) {
    ans.push_back(A[dq.front()]);
}
else
{
    ans.push_back(0);
}

//now process for remaining windows
for(int i = K; i<N; i++) {
    //first pop out of window element


    if(!dq.empty() && (i - dq.front())>=K ) {
        dq.pop_front();
    }

    //then push current element
    if(A[i] < 0)
       dq.push_back(i);

   // put in ans
   if(dq.size() > 0) {
        ans.push_back(A[dq.front()]);
   }
   else
   {
       ans.push_back(0);
   }
}
return ans;
```

easier

```cpp
vector<int> firstNegInt(vector<int>& arr, int k) {
    queue<int> q;          // store indices of negative numbers
    vector<int> ans;

    int n = arr.size();

    // process first window
    for(int i = 0; i < k; i++) {
        if(arr[i] < 0) q.push(i);
    }
    ans.push_back(q.empty() ? 0 : arr[q.front()]);

    // process remaining windows
    for(int i = k; i < n; i++) {
        // remove elements out of window
        while(!q.empty() && q.front() <= i - k) {
            q.pop();
        }

        // add current element if negative
        if(arr[i] < 0) q.push(i);

        // answer for this window
        ans.push_back(q.empty() ? 0 : arr[q.front()]);
    }

    return ans;
```

that's why storing index

First non repeating chars in string

ip → a a b c

op → a # b b

in string
"a" first non repeating char is a

"aa" first non repeating char not available because a is repeating

"aab" first non repeating char is b because a is repeating

"aabc" first non repeating char is b because a is repeating and c is not repeating but it is second non repeating

ip = q a c

op = a # c

ip : z a b c z a

op : z z z z a b

ip = a a b c

→ frequency count

→ q.push (ch)

→ q.front → character

q.empty → #

repeating → pop

non repeating → print

① a ② a ③ b ④ c

a→1 a→2 a→2, b→1 a→2, b→1, c→1

a a repeating b b

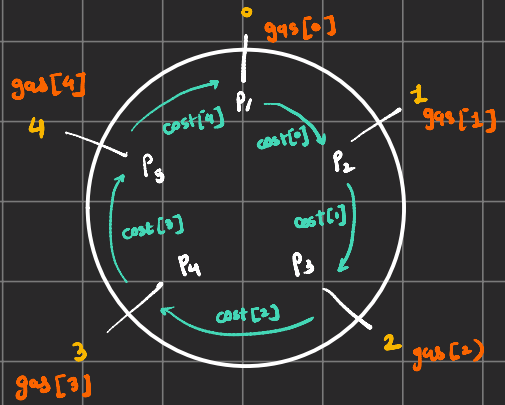
#

SC = O(26) + qsize O(n) = O(n)

TC = O(n) for (0→n) neglect → while (!empty)

```
string str = "aabc";
int freq[26] = {0};
queue<char> q;
string ans = "";

for(int i=0; i<str.length(); i++){
    char ch = str[i];

    // increment frequency
    freq[ch-'a']++;

    q.push(ch);

    while(!q.empty()){
        if(freq[q.front()-'a'] > 1){
            // repeating char
            q.pop();
        }else{
            ans.push_back(q.front());
            break;
        }
    }

    if(q.empty()){
        ans.push_back('#');
    }
}
cout << ans << endl;
```

Circular Tour / Gas station



assume 1 gas (liter) → cost (distance)

gas	1	2	3	4	5	
	0	1	2	3	4	index
cost	3	4	5	1	2	

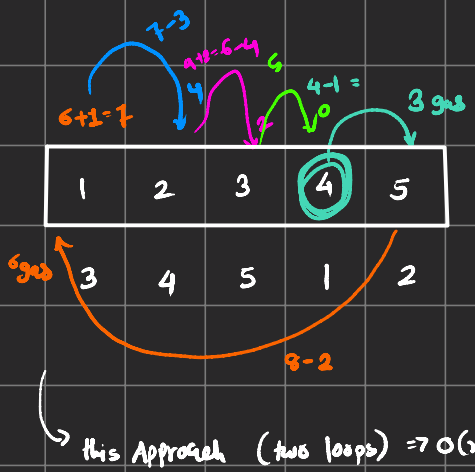
1 liter = 1 km distance

0th index : gas = 1, dist = 3 — 1 liter se 3 km nahi chal paege ✗

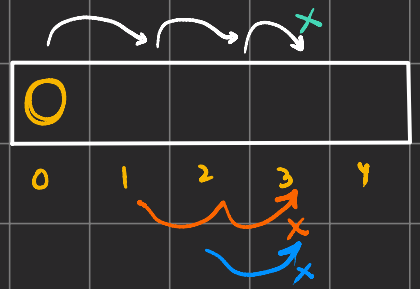
1th index : gas = 2, dist = 4 — 2 liter → 4 km ✗

2nd index : gas = 3, dist = 5 — 3 liter → 5 km ✗

3 index : gas = 4, dist = 1 — 4 liter → 1 km ✓



this Approach (two loops) => $O(n^2)$



just think that started from 0th index and then go to 1 → 2 → 3 (x) 3rd is not able to go bcz of cost

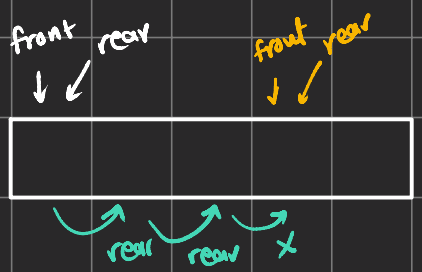
and now you are again going to try from 1st index

to try from 2nd index

Analysis: if you are not able to make it from 0th index where you already have some gas and the another gas is going to add when you visit station and failed inbetween so in this situation,

1st
2nd } x for both this is not going to work

optimization



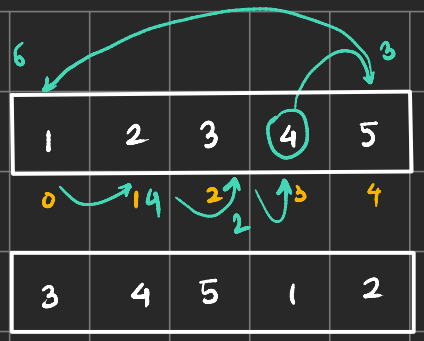
additional kaam bach gaya idhar because 1st and 2nd index ko skip karoge

movement possible → rear ++

movement not possible → front = rear + 1
rear = front

movement possible $P_x \rightarrow P_y$

(gas - dist >= 0)



0th index = P=1 → P-D = -27=0 X
D=3

1 index = P=2 → 2-4 = -2 X
D=4

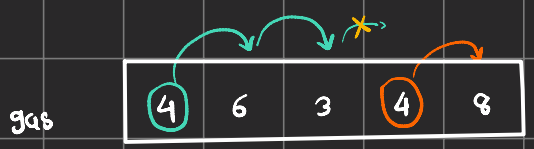
2 index = P=3 → 3-5 = -2 X
D=5

3 index = P=4 → 4-1 = 3 ✓ → P=3+5=8 D=2 → 8-2 = 6 ✓ → P=6+1=7 D=3 → 7-3 = 4 ✓

P=4+2=6 D=4 → 6-4=2 ✓ → P=2+3=5 D=5 → 5-5=0 ✓

go back to back so optimization showed above

now reached to index 3 circle complete



0th index : P = 4 > 1 → 7 > 1 → 4 → -3
D = 3 6 7 X deficit = -3

3rd index : P = 4 > 3 → 11 > 8 → remaining balance = 8
D = 1 3

if (balance >= deficit)
↳ circle complete ho jayega

TC: O(n)
SC: O(1)

```
int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
    int deficit = 0; // kitna gas kam pad raha he
    int balance = 0; // kitna gas bacha hua he
    int start = 0; // starting index

    for(int i=0; i<gas.size(); i++){
        balance = balance + gas[i] - cost[i];   // updated balance
        if(balance<0){
            deficit = deficit+balance;   ← galti hogi
            start = i+1;
            balance = 0;
        }
    }

    if(deficit+balance>=0){
        return start;
    }else{
        return -1;
    }
```

Sliding window maximum

- queue create
- first k element window process + answer
- remaining windows
  - answer
  - remove out of window
  - add new ele

nums: | 1 | 3 | -1 | -3 | 5 | 3 | 6 | 7 |

k = 3

①  q

② First k ele process (1, 3, -1)

q

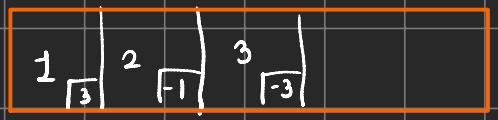
- assume 0th index biggest ele
- for 1st index arr[0] < arr[1] → so remove 1 and put 3
  1 < 3
- for 2nd index 3 > -1 → -1 ko 3 ko nahi hata sakta but ho sakta hai ki upcoming queues mei ko ans ho sakta hai
  → actually index storing

q

③

1	3	-1	-3	5	3	6	7
0	1	2	3	4	5	6	7

i



dq

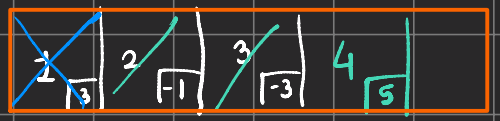
→ out of window (nothing here)

→ Add new ele - 3 → logic

+ Answer → q.front = arr[1]

3

{3, 3}



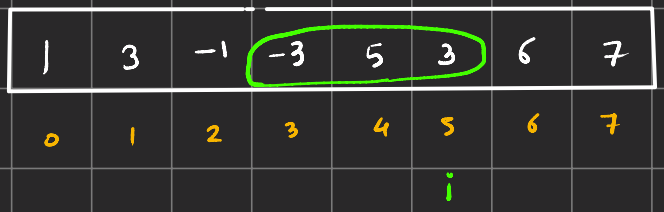
- out of window
  Here 1 (3)

- Inserting new ele
  Here 5 → logic → 5 se chote sub ko remove kar do

logic

```cpp
vector<int> maxSlidingWindow(vector<int>& nums, int k) {
    deque<int> dq;
    vector<int> ans;

    // first window
    for(int i=0; i<k; i++){
        // chote element delete kar do
        while(!dq.empty() && nums[i]>=nums[dq.back()]){
            dq.pop_back();
        }
        // insering index so we can check out of window element
        dq.push_back(i);
    }

    // store ans of first window
    ans.push_back(nums[dq.front()]);

    // remaining windows process
    for(int i=k; i<nums.size(); i++){
        // out of window ele ko remove
        if(!dq.empty() && i-dq.front() >= k){
            dq.pop_front();
        }

        // ab firse current ele k liye chote ele ko remove karna he
        while(!dq.empty() && nums[i]>=nums[dq.back()]){
            dq.pop_back();
        }
        // insering index so we can check out of window element
        dq.push_back(i);

        // current window ka ans store karna
        ans.push_back(nums[dq.front()]);
    }

    return ans;
```

{3,3,5}



→ Remove out of Bound
  Here nothing

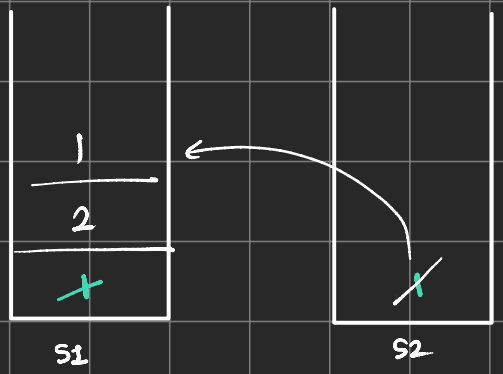
→ Add new Ele Here 3 → 3 se chota koi nahi
  → 3 se bada 5 hai
    So 3 ko push karda

{3,3,5,5}
- - - do for all windows

* Queue using Stack

FIFO LIFO

(M1) Push → O(n)



① $S_1 \rightarrow S_2$ (push)
② add x to S_1
③ $S_2 \rightarrow S_1$ (push)

push(1)
push(2)

* q.front() → O(1) = s1.top()

* q.pop() → O(1) = s1.pop()

(M2)

push(x) → O(1)

push(x) → s1

pop() → O(n)

if (s2 not empty) → s2.pop()

else → s1 → s2
s2.pop()

(s1 empty ho jayega idhar)

front() → O(n)

if (s2 not empty) → s2.top()

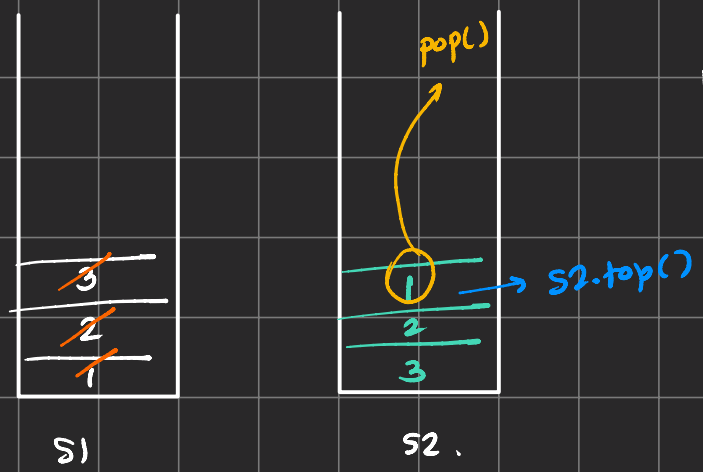
else → s1 → s2
s2.top()

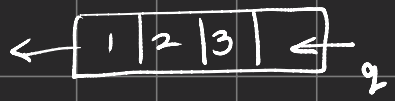
push(1)
push(2)
push(3)

push() → 1,2,3 → s1

front() → s1 → s2
s2.top

pop() → s2 isnot empty now → s2.pop()






```cpp
class MyQueue {
    stack<int>s1,s2;
public:
    MyQueue() {
    }

    void push(int x) {
        s1.push(x);
    }

    int pop() {
        int pop=-1;
        if(!s2.empty()){
            pop=s2.top();
        }else{
            while(!s1.empty()){
                s2.push(s1.top());
                s1.pop();
            }
            pop = s2.top();
        }
        s2.pop();
        return pop;
    }

    int peek() {
        int front = -1;
        if(!s2.empty()){
            front = s2.top();
        }else{
            while(!s1.empty()){
                s2.push(s1.top());
                s1.pop();
            }
            front = s2.top();
        }
        return front;
    }

    bool empty() {
        return s1.empty() && s2.empty();
    }
};
```

Stacks using queues

(M1) push(x)

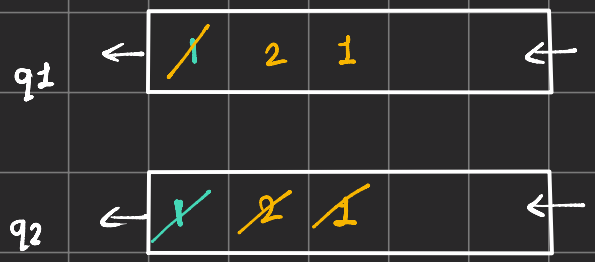
→ push x to Q_2 (1)

→ $Q_1 \rightarrow Q_2$

→ $Q_2 \rightarrow Q_1$

pop() → q1.front() will give you top of Stack

top() →



→ 1 to q2

→ $Q_1 \rightarrow Q_2$ (not here)

→ $Q_2 \rightarrow Q_1$

→ 2 to q2

→ Q_1 to Q_2

→ Q_2 to Q_1

(M2) Stack using only one queue

push(1)

push(2)

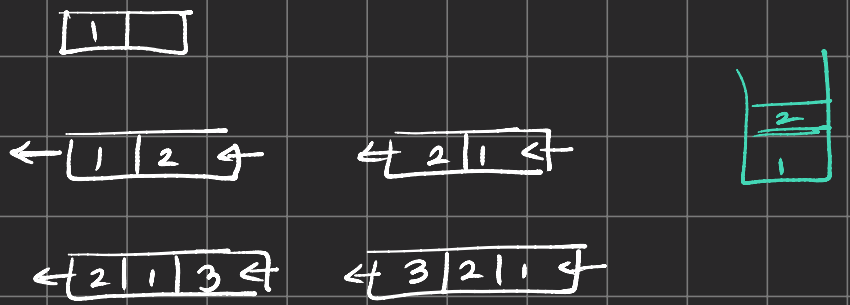
push(3)

- push into queue
- loop (size-1) → x = q.pop()
  → q.push(x)

push(1)

push(2)

push(3)



```cpp
class MyStack {
    queue<int> q;
public:
    MyStack() {

    }

    void push(int x) {
        q.push(x);
        for(int i=0; i<q.size()-1; i++){
            int front = q.front();
            q.pop();
            q.push(front);
        }

    }

    int pop() {
        int top = q.front();
        q.pop();
        return top;
    }

    int top() {
        int top = q.front();
        return top;
    }

    bool empty() {
        return q.empty();
    }
};
```

170 → remainings ⊕ (k stacks in array)

171 → also remains