Queues are used for FIFO operations

eg → first negative element in window :

brute → (Scan every window) ⇒ {-8, 2, 3, -6, 10}

0	1	2	3	4
-8	2	3	-6	10

```
→ vector<int> ans;
  int n = arr.size();
  for(int i=0; i<=n-k; i++)
  {
     for(int j=i, j<i+k; j++) } Scaning (k) size
     {
        if(arr[j]<0) //negative element
        { ans.push_back (arr[j]);
          break;
        }
     }
   if(j==i+k) // koi negative ni
   { ans.push_back (0);
   }
  }
  return ans
```

i = 0;
j = 0;

(5 - 3)
→ (2)

-8	-6	

queue will reduce the time to (O(k)

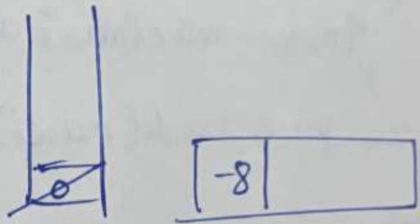
1) fixed size SW;
2) include element
3) window ready (i>=k-1)
4) remove out of window
5) compute answer

```
queue<int> q;
v<int> ans;
for(int i=0;i<arr.size(); i++)
{ if (arr[i]<0)
   { q.push(i);
   }
  if(i >= k-1) //window end;
  { while (!q.empty && q.front() < i-k+1)
     { q.pop();
     }
     if (q.empty) push 0;
     else ans.push (arr(q.front) );
  }
}
```

① if arr(neg) push index to queue;

```
if(arr[i]<0)
{ q.push(i);
}
if(i>=k-1)
{ while(!q.empty() && q.front()<i-k+1)   // mtlb previous window remov
  { q.pop();
  }
  if(q.empty())
  { ans.push_back(0);
  }
  else{
    ans.push_back(arr[q.front()]);
  }
}
return ans;
}
```

arr → {-[8, 2] 3, -6, 10}
0 ① 2 3 4



k=② ⇒ if(i>=k+1)
{ i=①. i=②. i=③.
(0 < 1-2+1 0)

M= [8,2,3.-6,10] ans 
k=2;

if(i<i+k+1)

K Sized Subarray maximum:

har K sized window mai maximum K track Krna hai or store Krna hai;

Brute sochu agr:

har bar i → k

```
for(int i=0; i<N-K; i++)
{
  int maxi=INT_MIN;
  int j;
  for(int j=i; j<i+K; j++)
  {
    maxi = max(maxi, arr[j]);
  }
  ans.push_back(maxi);
}
return ans;
}
```

not applicable for largest /large input; O(n²).

→ Use dequeue;

```
deque<int> q;
vector<int> ans;
for(i → n)
{
  while(!q.empty() && arr[q.front()] <= arr[i])
  {
    q.pop_back();              // LIFO maintain
  }
  q.push_back(i);             // insert
  if(q.front() < i-k+1)       // out of boundry of k
  {
    q.pop_front();            // front s pop()
  }
  if(i<=k-1) ans.push_back(arr[q.front()]);
}
return ans;
}
```

Circular Deque:

insertion / deletion from both side

~~dequeue~~ deque has → push_front() → extra
push_back() } normal
pop_front()
~~pop~~ pop_back() → extra

Deque implement hota hai circular array pe.

→ space wastage ×
→ front/rear O(1)
→ Queue full bhi hoti ~~hai~~, empty bhi hoti hai.

eg → size = 5

f=0, r=0 [10 - - - -] , f=0, r=1
push_back(10)
push_back(20) [10, 20, - - -], f=0, r=2
pop_front()
[- 20 - - -] f=1, r=1
push_back(30) → [- 20 30 - -] f=1, r=2
push_back(40) → [- 20 30 40 -] f=1, r=3
push_back(50) [- 20, 30, 40 50] f=1, r=4
push_back(60) [60, 20, 30, 40, 50] f=1, r=0

$r = (r+1) \% 5 \Rightarrow 5 \% 5 = 0$

empty condition
$\Rightarrow (f == -1)$

full $\Rightarrow ((r+1) \% size == r)$

{ is used to efficiently utilize space and perform insertion & deletion from both ends in O(1) }

Circular deque = deque + modulo (%)

```
public:
  int *arr;
  int size, front, rear;
  MyCdeque(int capacity)
     size = capacity
     arr = new int[size];
     f = r = -1;
  }
```

① insertfromfront(int x)

```
{
 if (isFull()) return false;
 else { if (isEmpty())
     {
       f = r = 0;
     }
 elseif
   f = (f - 1 + size) % size;
 }
 a[f] = value; x;
```

② bool atlast(x)

```
{
 if (is ~~full~~) return false;
 if (isEmpty()) r = f = 0;
 else {
   r = (r+1) % size();
   arr[r] = x;
 }
 return true,
```

③ deletefront ()

```
{ if ( isEmpty() return false;
 if (r == f)
   r = f = -1;
 else {
   f = (f +1) % size;
 }
}
```

④ Deletelast ()

```
 if (isEmpty()) return false;
 if (r == f)  r = f = -1;
 else {
   r = (r - 1 + size) % size;
 }
 return true;
```

next = (index +1) % size

prev = (index - 1) % size

for → reversal / undo / Nested → Stack

order / Scheduling / stream → queue

Window / max / min / cache → dequeue

TRAPS

① Difference b/w Stacks & Queue?

Ans) Stacks are used when recent element matters (undo, reverse)

→ Queue follows when order/fairness matters (scheduling)

② when to not use stacks (TRAP)?

→ Scheduling / buffering

③ sliding window (Q/S)?

→ deque, cause we need to keep adding & removing from the window from both side, so stacks are not used.

④ Reverse the datastream?

→ Use Stacks

cause queues preserves the order

⑤ BFS uses stack or Queues?

→ Uses queue

cause we need to maintain level by level order.

⑥ UNDO / REDO ? stacks

⑦ producer/Consumer problem?

→ Queues

⑧ Cache implementations (Deques & hashmaps)

cause → removes last used

→ adds recent one

⑨ Can queue be implemented by using stack?

→ Yes, can be implemented using 2 stacks

One for input / output

Amortized O(1)

⑩ Needs middle element?

→ NO stacks/queue.

cause we need Deque / DLL.

① fixed window (Basic) :
→ subarray size k
→ first k / last k

(Queue / variables)

② fixed window (min / max) → since remove one get one :
→ dequeue

③ fixed window + condition (count / property) :
→ First (FIFO)
→ queue / hashmaps

④ Variable Size window (Sum) / Length :
→ 2 pointers

⑤ Variable window + strings (longest substring) :
→ hashmaps / fixed window

⑥ Anagram / permutation :
→ hashmap / fixed window

⑦ Sliding window on circular array
→ Two pointer + mod

⑧ Stream based → queue

⑨ Sliding window + Greedy (deque) / (Stack)

⑩ Two ~~pointers~~ window
→ Atmost trick.

★ (Sliding window + mod)

→ array ko 2 times imagine karo

temp = [8, 3, 1, 2, 8, 3, 1, 2]

→ Ab sliding window laga do

→ window length = k

→ But window start index < n hi allow karo.

```
vector<int> temp;
for (int i=0; i<2*n; i++)
{
    temp.push_back(arr[i%n]);
// normal window of size k
```

LRU Cache :

LRU(4) ✓ size of cache discussed.

put(2,6)   (7,10) (8,11) (4,7) ~~(2,6)~~

put(4,7)

put(8,11)   // 6.   (2,6) (7,10) (8,11) (4,7)

put(7,10)

$\boxed{\text{get(2)}}$ (8,11) (2,6) (7,10) (4,7)

(get(8))   // 11

put(5,6)   //   (5,6) (8,11) (2,6) (7,10) (4,7)

get(7)   // 10   (7,10) (5,6) (8,11) (2,6) (7,10) (4,7)

put(5,7)   (5,7) (7,10) (5,6) (8,11) (2,6) (7,10) (4,7)

ans //.