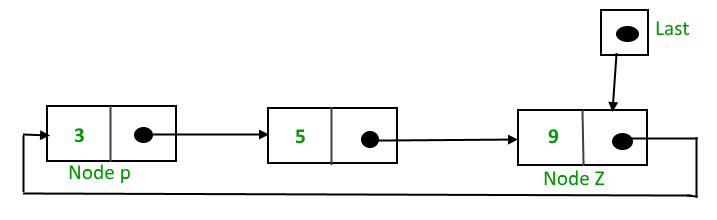
**Topic 1 : Circular Linked Lists**

*A****circular linked****list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.*

Below is a pictorial representation of Circular Linked List:  
  
  
**Implementation**:  
To implement a circular singly linked list, we take an external pointer that points to the last node of the list. If we have a pointer last pointing to the last node, then *last -> next* will point to the first node.  
  
  
The pointer ***last***points to node **Z**and ***last -> next*** points to the node **P**.

**Why have we taken a pointer that points to the last node instead of the first node?**

For insertion of nodes in the beginning we need to traverse the whole list. Also, for insertion and the end, the whole list has to be traversed. If instead of a start pointer we take a pointer to the last node then in both the cases there won’t be any need to traverse the whole list. So insertion in the beginning or at the end takes constant time irrespective of the length of the list.  
  
Below is a sample program to create and traverse in a Circular Linked List in both Java and C++:

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// A complete C++ program to demonstrate the

// working of Circular Linked Lists

#include<bits/stdc++.h>

using namespace std;

// Circular Linked List Node

struct Node

{

int data;

struct Node \*next;

};

// Function to add a node at the end of a

// Circular Linked List

struct Node \*addEnd(struct Node \*last, int data)

{

if (last == NULL)

{

// Creating a node dynamically.

struct Node \*temp = new Node;

// Assigning the data.

temp -> data = data;

last = temp;

// Creating the link.

last -> next = last;

return last;

Run

Java



**Output**:

6 4 2 8 12 10

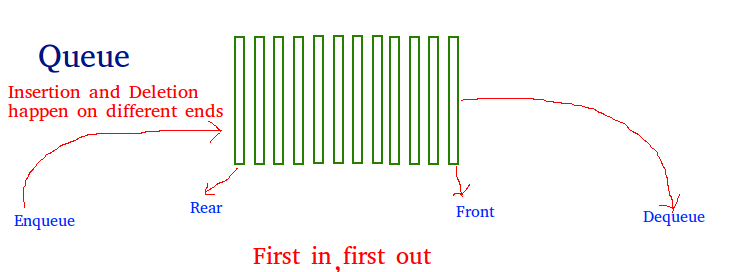
**Advantages of Circular Linked Lists:**

1. Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
2. Useful for implementation of a queue. Unlike this implementation, we don’t need to maintain two pointers for front and rear if we use a circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.
3. Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.
4. Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap.

**Topic 2 : Introduction to Queues**

Like the Stackdata structure, ***Queue***is also a linear data structure which follows a particular order in which the operations are performed. The order is **F**irst **I**n **F**irst **O**ut (FIFO) which means that the element which is inserted first in the queue will be the first one to be removed from the queue. A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first.  
  
The difference between stacks and queues is in removing. In a stack we remove the most recently added item; in a queue, we remove the least recently added item.  
  
**Operations on Queue:** Mainly the following four basic operations are performed on queue:

* **Enqueue:**Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
* **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.
* **Front:**Get the front item from the queue.
* **Rear:** Get the last item from the queue.

  
  
  
**Array implementation Of Queue**: For implementing a *queue*, we need to keep track of two indices, front and rear. We enqueue an item at the rear and dequeue an item from the front. If we simply increment front and rear indices, then there may be problems, the front may reach the end of the array. The solution to this problem is to increase front and rear in a circular manner.  
  
Consider an Array of size **N**is taken to implement a queue. Initially, the size of the queue will be zero(0). The total capacity of the queue will be the size of the array i.e. N. Now initially, the index *front*will be equal to 0, and *rear*will be equal to N-1. Every time on inserting an item, the index *rear*will increment by one, so increment it as: **rear = (rear + 1)%N** and every time on removing an item, the front index will shift to right by 1 place so increment it as: **front = (front + 1)%N**.  
  
**Example**:

*Array = queue[N].  
front = 0, rear = N-1.  
N = 5.*  
  
**Operation 1**:  
*enque(5);*  
front = 0,   
rear = (N-1 + 1)%N = 0.  
Queue contains: [5].  
  
**Operation 2**:  
*enque(10);*  
front = 0,   
rear = (rear + 1)%N = (0 + 1)%N = 1.  
Queue contains: [5, 10].  
  
**Operation 3**:  
*enque(15);*  
front = 0,   
rear = (rear + 1)%N = (1 + 1)%N = 2.  
Queue contains: [5, 10, 15].  
  
**Operation 4**:  
*deque();*  
print queue[front];   
front = (front + 1)%N = (0 + 1)%N = 1.  
Queue contains: [10, 15].

Below is the Array implementation of queue in C++ and Java:

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// CPP program for array implementation of queue

#include <bits/stdc++.h>

using namespace std;

// A structure to represent a queue

class Queue

{

public:

int front, rear, size;

unsigned capacity;

int\* array;

};

// function to create a queue of given capacity.

// It initializes size of queue as 0

Queue\* createQueue(unsigned capacity)

{

Queue\* queue = new Queue();

queue->capacity = capacity;

queue->front = queue->size = 0;

queue->rear = capacity - 1; // This is important, see the enqueue

queue->array = new int[(queue->capacity \* sizeof(int))];

return queue;

}

// Queue is full when size

// becomes equal to the capacity

int isFull(Queue\* queue)

{ return (queue->size == queue->capacity); }

Run

Java



Output:

10 enqueued to queue

20 enqueued to queue

30 enqueued to queue

40 enqueued to queue

10 dequeued from queue

Front item is 20

Rear item is 40

**Time Complexity:** Time complexity of all operations like enqueue(), dequeue(), isFull(), isEmpty(), front() and rear() is O(1). There is no loop in any of the operations.  
  
**Applications of Queue:** Queue is used when things don’t have to be processed immediately, but have to be processed in **F**irst **I**n**F**irst **O**ut order like Breadth First Search. This property of Queue makes it also useful in following kinds of scenarios.

1. When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
2. When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

**Topic 3 : Implementing Queue in C++ and Java using Built-in Classes**

Queue in C++ STL

The Standard template Library in C++ offers a built-in implementation of the Queue data structure for simpler and easy use. The STL implementation of queue data structure implements all basic operations on queues such as enque(), deque(), clear() etc.  
  
**Syntax**:

*queue<* ***data\_type*** *>* ***queue\_name****;*  
  
where,  
**data\_type** is the type of element to be stored   
in the queue.  
**queue\_name** is the name of the queue data structure.

**The functions supported by std::queue are**:

* **empty()** – Returns whether the queue is empty.
* **size()** – Returns the size of the queue.
* **swap()**: Exchange the contents of two queues but the queues must be of the same type, although sizes may differ.
* **emplace()**: Insert a new element into the queue container, the new element is added to the end of the queue.
* **front() and back()**: front() function returns a reference to the first element of the queue. back() function returns a reference to the last element of the queue.
* **push(g) and pop()**: The push() function adds the element ‘g’ at the end of the queue. The pop() function deletes the first element of the queue.



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// CPP code to illustrate

// Queue in Standard Template Library (STL)

#include <iostream>

#include <queue>

using namespace std;

void showq(queue <int> gq)

{

queue <int> g = gq;

while (!g.empty())

{

cout << '\t' << g.front();

g.pop();

}

cout << '\n';

}

int main()

{

queue <int> gquiz;

gquiz.push(10);

gquiz.push(20);

gquiz.push(30);

cout << "The queue gquiz is : ";

showq(gquiz);

cout << "\ngquiz.size() : " << gquiz.size();

cout << "\ngquiz.front() : " << gquiz.front();

Run

**Output**:

The queue gquiz is : 10 20 30  
  
gquiz.size() : 3  
gquiz.front() : 10  
gquiz.back() : 30  
gquiz.pop() : 20 30

Queue interface in Java

The Queue interface is available in the java.util package and extends the Collection interface. The queue collection is used to hold the elements about to be processed and provides various operations like the insertion, removal etc. It is an ordered list of objects with its use limited to insert elements at the end of the list and deleting elements from the start of list i.e. it follows the FIFO or the First-In-First-Out principle. Being an interface the queue needs a concrete class for the declaration and the most commonly used classes are the PriorityQueue and LinkedList in Java. It is to be noted that both the implementations are not thread safe. PriorityBlockingQueue is one alternative implementation if thread safe implementation is needed.  
  
**Methods in Queue**:

* **add()**- This method is used to add elements at the tail of the queue. More specifically, at the last of linked list if it is used, or according to the priority in case of priority queue implementation.
* **peek()**- This method is used to view the head of a queue without removing it. It returns Null if the queue is empty.
* **element()**- This method is similar to peek(). It throws NoSuchElementException when the queue is empty.
* **remove()**- This method removes and returns the head of the queue. It throws NoSuchElementException when the queue is empty.
* **poll()**- This method removes and returns the head of the queue. It returns null if the queue is empty.
* **size()**- This method returns the no. of elements in the queue.

Below is a simple Java program to demonstrate these methods:



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// Java orogram to demonstrate working of Queue

// interface in Java

import java.util.LinkedList;

import java.util.Queue;

public class QueueExample

{

public static void main(String[] args)

{

Queue<Integer> q = new LinkedList<>();

// Adds elements {0, 1, 2, 3, 4} to queue

for (int i=0; i<5; i++)

q.add(i);

// Display contents of the queue.

System.out.println("Elements of queue-"+q);

// To remove the head of queue.

int removedele = q.remove();

System.out.println("removed element-" + removedele);

System.out.println(q);

// To view the head of queue

int head = q.peek();

System.out.println("head of queue-" + head);

// Rest all methods of collection interface,

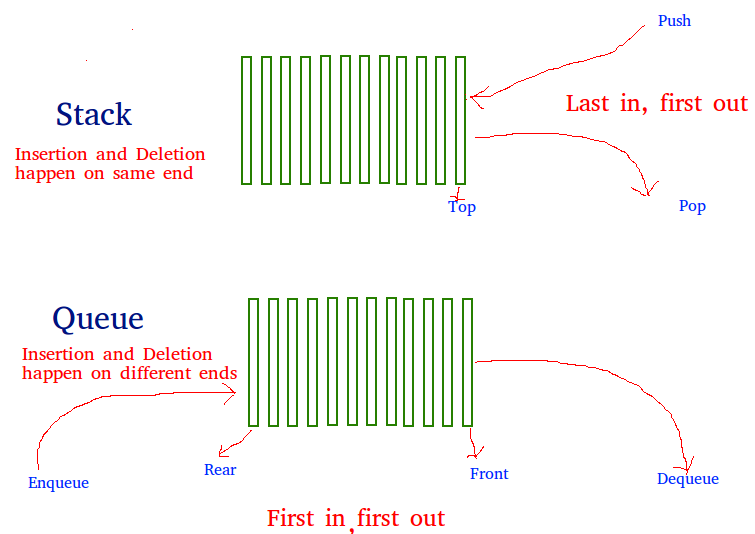
// Like size and contains can be used with this

Run

**Output**:

Elements of queue-[0, 1, 2, 3, 4]  
removed element-0  
[1, 2, 3, 4]  
head of queue-1  
Size of queue-4

**Topic 4 : Implementing Queue using Stack**

**Problem**: Given a stack data structure with push and pop operations, the task is to implement a queue using instances of stack data structure and operations on them.  
  
  
  
  
**Solution**: *A queue can be implemented using two stacks.* Let the queue to be implemented be **q**and stacks used to implement **q**are **stack1**and **stack2** respectively.  
  
The queue ***q*** can be implemented in two ways:

* **Method 1 (By making enQueue operation costly)**: This method makes sure that the oldest entered element(element inserted first) is always at the top of stack1, so that deQueue operation just pops from stack1. To put the element at top of stack1, stack2 is used. The idea is to, while pushing an element, first move all elements from stack1 to stack2, insert the new element to stack1 and then again move all elements from stack2 to stack1.  
    
  Below is the implementation of both enQueue() and deQueue() operations:

**enQueue(q, x)**  
 1) While stack1 is not empty, push everything from stack1 to stack2.  
 2) Push x to stack1 (assuming size of stacks is unlimited).  
 3) Push everything back to stack1.  
Here the time complexity will be O(n)  
  
**deQueue(q)**  
 1) If stack1 is empty then print an error  
 2) Pop an item from stack1 and return it  
Here time complexity will be O(1)

**Implementation**:

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// CPP program to implement Queue using

// two stacks with costly enQueue()

#include <bits/stdc++.h>

using namespace std;

struct Queue {

stack<int> s1, s2;

void enQueue(int x)

{

// Move all elements from s1 to s2

while (!s1.empty()) {

s2.push(s1.top());

s1.pop();

}

// Push item into s1

s1.push(x);

// Push everything back to s1

while (!s2.empty()) {

s1.push(s2.top());

s2.pop();

}

}

// Dequeue an item from the queue

int deQueue()

{

// if first stack is empty

Run

Java



**Output:**

1

2

3

* **Method 2 (By making deQueue operation costly)**: In this method, in en-queue operation, the new element is entered at the top of stack1. In de-queue operation, if stack2 is empty then all the elements are moved to stack2 and finally, the top of stack2 is returned.  
    
  Below is the implementation of both enQueue() and deQueue() operations:

**enQueue(q, x)**  
 1) Push x to stack1 (assuming size of stacks is unlimited).  
Here time complexity will be O(1)  
  
**deQueue(q)**  
 1) If both stacks are empty then error.  
 2) If stack2 is empty  
 While stack1 is not empty, push everything from stack1 to stack2.  
 3) Pop the element from stack2 and return it.  
Here time complexity will be O(n)

*Method 2 is better in performance than method 1.*As Method 1 moves all the elements twice in enQueue operation, while method 2 (in deQueue operation) moves the elements once and moves elements only if stack2 is empty.  
  
**Implementation**:

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// CPP program to implement Queue using

// two stacks with costly deQueue()

#include <bits/stdc++.h>

using namespace std;

struct Queue {

stack<int> s1, s2;

// Enqueue an item to the queue

void enQueue(int x)

{

// Push item into the first stack

s1.push(x);

}

// Dequeue an item from the queue

int deQueue()

{

// if both stacks are empty

if (s1.empty() && s2.empty()) {

cout << "Q is empty";

exit(0);

}

// if s2 is empty, move

// elements from s1

if (s2.empty()) {

while (!s1.empty()) {

s2.push(s1.top());

s1.pop();

Run

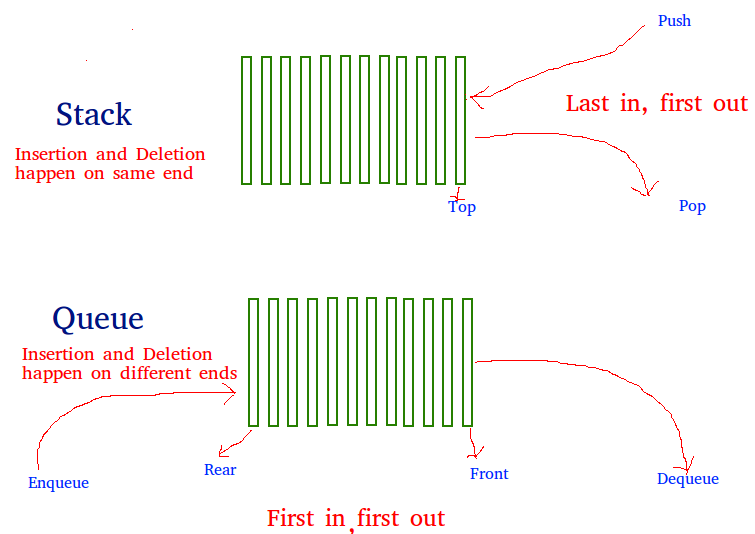
Java



**Output:**

1 2 3

**Topic 5 : Implementing Stack Using Queue**

**Problem**: Given a Queue data structure that supports standard operations like enqueue() and dequeue(). We need to implement a Stack data structure using only instances of Queue and queue operations allowed on the instances.  
  
  
  
This problem is just the opposite of the problem described in the previous post of implementing a queue using stacks. Similar to the previous problem, a **stack**can also be implemented using two queues. Let stack to be implemented be **'s'** and queues used to implement be **'q1'** and **'q2'**.  
  
Stack 's' can be implemented in two ways:

* **Method 1 (By making push operation costly)**: This method makes sure that newly entered element is always at the front of 'q1', so that pop operation just dequeues from 'q1'. The queue, 'q2' is used to put every new element in front of 'q1'.

**push(s, x)** // x is the element to be pushed and s is stack  
 1) Enqueue x to q2  
 2) One by one dequeue everything from q1 and enqueue to q2.  
 3) Swap the names of q1 and q2   
// Swapping of names is done to avoid one more   
// movement of all elements from q2 to q1.   
  
**pop(s)**  
 1) Dequeue an item from q1 and return it.

**Implementation**:

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// C++ program to implement a stack using

// two queues

#include<bits/stdc++.h>

using namespace std;

// Stack class

class Stack

{

// Two inbuilt queues

queue<int> q1, q2;

// To maintain current number of

// elements

int curr\_size;

public:

Stack()

{

curr\_size = 0;

}

// Function to implement push() operation

void push(int x)

{

curr\_size++;

// Push x first in empty q2

q2.push(x);

Run

Java



**Output :**

current size: 3  
3  
2  
1  
current size: 1

* **Method 2 (By making pop operation costly)**: In push operation, the new element is always enqueued to q1. In pop() operation, if q2 is empty then all the elements except the last, are moved to q2. Finally the last element is dequeued from q1 and returned.

**push(s, x)**  
 1) Enqueue x to q1 (assuming size of q1 is unlimited).  
  
**pop(s)**   
 1) One by one dequeue everything except the last element   
 from q1 and enqueue to q2.  
 2) Dequeue the last item of q1, the dequeued item   
 is the result, store it.  
 3) Swap the names of q1 and q2  
 4) Return the item stored in step 2.  
// Swapping of names is done to avoid one more  
// movement of all elements from q2 to q1.

**Implementation**:



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// Program to implement a stack using two queues

#include<bits/stdc++.h>

using namespace std;

// Stack class

class Stack

{

queue<int> q1, q2;

int curr\_size;

public:

Stack()

{

curr\_size = 0;

}

void pop()

{

if (q1.empty())

return;

// Leave one element in q1 and

// push others in q2.

while (q1.size() != 1)

{

q2.push(q1.front());

q1.pop();

}

Run

**Output :**

current size: 4  
4  
3  
2  
current size: 2

**Topic 6 : Sample Problems on Queue**

**Problem #1 : Reversing the first K elements of a Queue**

**Description -** Given an integer k and a queue of integers, we need to reverse the order of the first k elements of the queue, leaving the other elements in the same relative order.  
Only the following standard operations are allowed on the queue.

* enqueue(x) : Add an item x to rear of queue
* dequeue() : Remove an item from front of queue
* size(( : Returns number of elements in queue.
* front() : Finds front item.

Input : Q = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]  
 k = 5  
Output : Q = [50, 40, 30, 20, 10, 60, 70, 80, 90, 100]

**Solution -**The idea is to use an auxiliary stack and follow these steps to solve the problem -

1. Create an empty stack.
2. One by one dequeue items from a given queue and push the dequeued items to stack.
3. Enqueue the contents of stack at the back of the queue.
4. Reverse the whole queue.

**Pseudo Code**

/\* Function to reverse the first K elements of the Queue \*/  
void reverseQueueFirstKElements(k, Queue)   
{   
 if (Queue.empty() == true || k > Queue.size())   
 return  
 if (k <= 0)   
 return  
 stack Stack   
 /\* Push the first K elements into a Stack\*/  
 for ( i = 1 to k) {   
 Stack.push(Queue.front())  
 Queue.pop()  
 }   
 /\* Enqueue the contents of stack   
 at the back of the queue\*/  
 while (!Stack.empty()) {   
 Queue.push(Stack.top())  
 Stack.pop()  
 }   
 /\* Remove the remaining elements and   
 enqueue them at the end of the Queue\*/  
 for (int i = 0 to i < Queue.size() - k) {   
 Queue.push(Queue.front())  
 Queue.pop()  
 }   
}

**Time Complexity :** O(n) , n : size of queue  
**Auxiliary Space :** O(k)

**Problem #2 : Sliding Window Maximum**

**Description -** Given an array and an integer k, find the maximum for each and every contiguous subarray of size k.

Input :  
arr[] = {1, 2, 3, 1, 4, 5, 2, 3, 6}  
k = 3   
Output :  
3 3 4 5 5 5 6

**Solution :** We create a Deque, Qi of capacity k, that stores only useful elements of the current window of k elements. An element is useful if it is in the current window and is greater than all other elements on the left side of it in the current window. We process all array elements one by one and maintain Qi to contain useful elements of the current window and these useful elements are maintained in sorted order. The element at front of the Qi is the largest and the element at the rear of Qi is the smallest of the current window.

void printKMax(arr[], n, k)   
{   
 // Create a Double Ended Queue, Qi that will store indexes of array elements   
 // The queue will store indexes of useful elements in every window and it will   
 // maintain decreasing order of values from front to rear in Qi, i.e.,   
 // arr[Qi.front[]] to arr[Qi.rear()] are sorted in decreasing order   
 deque < int > Qi(k)  
   
 /\* Process first k (or first window) elements of array \*/  
 for (i = 0; i < k; ++i) {   
 // For every element, the previous smaller elements are useless so   
 // remove them from Qi   
 while ((!Qi.empty()) && arr[i] >= arr[Qi.back()])   
 Qi.pop\_back() // Remove from rear   
   
 // Add new element at rear of queue   
 Qi.push\_back(i)  
 }   
   
 // Process rest of the elements, i.e., from arr[k] to arr[n-1]   
 for (; i < n; ++i) {   
 // The element at the front of the queue is the largest element of   
 // previous window, so print it   
 print (arr[Qi.front()])   
   
 // Remove the elements which are out of this window   
 while ((!Qi.empty()) && Qi.front() <= i - k)   
 Qi.pop\_front() // Remove from front of queue   
   
 // Remove all elements smaller than the currently   
 // being added element (remove useless elements)   
 while ((!Qi.empty()) && arr[i] >= arr[Qi.back()])   
 Qi.pop\_back()  
   
 // Add current element at the rear of Qi   
 Qi.push\_back(i)  
 }   
   
 // Print the maximum element of last window   
 print (arr[Qi.front()])   
}