


[← Back To Course \(/batchPage.php?batchId=308\)](#) Practice(<https://practice.geeksforgeeks.org/home/>) Learn Quiz

Learn

Problems

Quiz

Filter

LIVE BATCHES

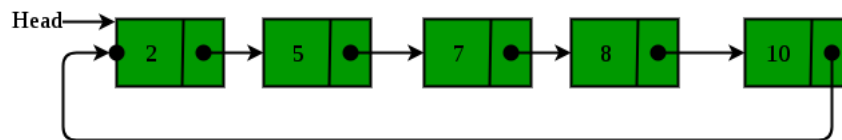
Learn

Circular Linked Lists

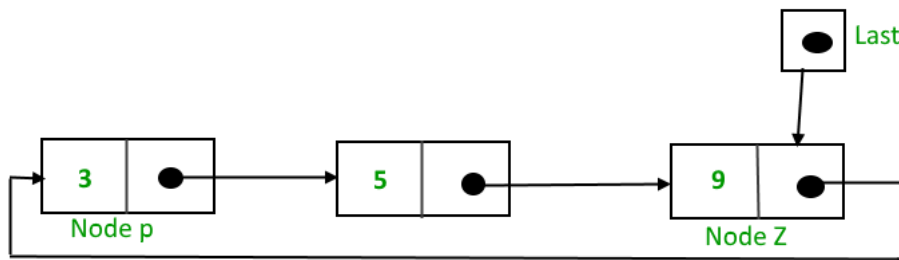


A **circular linked** list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.

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 first node?

For insertion of node in the beginning we need to traverse the whole list. Also, for insertion at the end, the whole list has to be traversed. If instead of 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 // A complete C++ program to demonstrate the
3 // working of Circular Linked Lists
4
5 #include<bits/stdc++.h>
6 using namespace std;
7
8 // Circular Linked List Node
```

```
9 struct Node
10 {
11     int data;
12     struct Node *next;
13 };
14
15 // Function to add a node at the end of a
16 // Circular Linked List
17 struct Node *addEnd(struct Node *last, int data)
18 {
19     if (last == NULL)
20     {
21         // Creating a node dynamically.
22         struct Node *temp = new Node;
23
24         // Assigning the data.
25         temp -> data = data;
26         last = temp;
27
28         // Creating the link.
29         last -> next = last;
30     }
```

[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 the front can always be obtained as the 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.

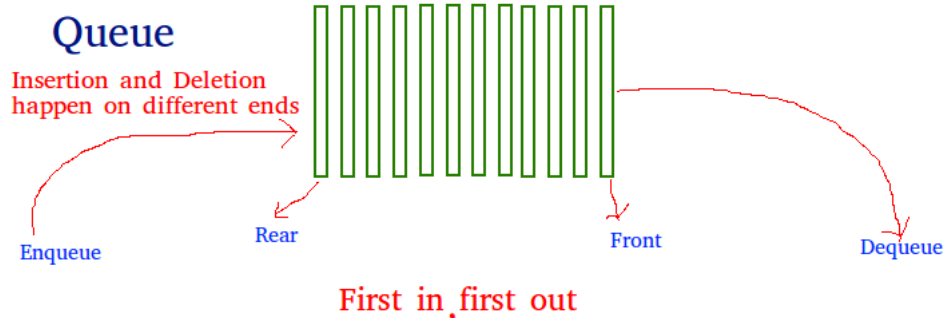
- Introduction to Queues

Like *Stack* data structure, **Queue** is also a linear data structure that follows a particular order in which the operations are performed. The order is **First In First Out (FIFO)**, which means that the element that is inserted first in the queue will be the first one to be removed from the queue. A good example of queue is any queue of consumers for a resource where the consumer who came first is served first.

The difference between stacks and queues is in removing. In a stack, we remove the most recently added item; whereas, 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 queue.
- **Rear:** Get the last item from queue.



(<https://media.geeksforgeeks.org/wp-content/cdn-uploads/gq/2014/02/Queue.png>)

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 that 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 an item is inserted, so the index *rear* will increment by one, hence increment it as: **rear = (rear + 1)%N** and everytime an item is removed, so the front index will shift to right by 1 place, hence increment it as: **front = (front + 1)%N**.

Example:

```
Array = queue[N].
front = 0, rear = N-1.
N = 5.
```

Operation 1:

```
enqueue(5);
front = 0,
rear = (N-1 + 1)%N = 0.
Queue contains: [5].
```

Operation 2:

```
enqueue(10);
front = 0,
rear = (rear + 1)%N = (0 + 1)%N = 1.
Queue contains: [5, 10].
```

Operation 3:

```
enqueue(15);
front = 0,
rear = (rear + 1)%N = (1 + 1)%N = 2.
Queue contains: [5, 10, 15].
```

Operation 4:

```
dequeue();
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 // CPP program for array implementation of queue
3 #include <bits/stdc++.h>
4 using namespace std;
5
6 // A structure to represent a queue
7 class Queue
8 {
9     public:
10     int front, rear, size;
11     unsigned capacity;
12     int* array;
13 };
```

```

14
15 // function to create a queue of a given capacity.
16 // It initializes the size of the queue as 0
17 Queue* createQueue(unsigned capacity)
18 {
19     Queue* queue = new Queue();
20     queue->capacity = capacity;
21     queue->front = queue->size = 0;
22     queue->rear = capacity - 1; // This is important, see the enqueue
23     queue->array = new int[(queue->capacity * sizeof(int))];
24     return queue;
25 }
26
27 // Queue is full when size
28 // becomes equal to the capacity
29 int isFull(Queue* queue)
30 { 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 such as 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 **First In First Out** order like Breadth First Search (http://en.wikipedia.org/wiki/Breadth-first_search). This property of Queue makes it also useful in following kind 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.

- 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 queue such as enqueue(), 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 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 // CPP code to illustrate
3 // Queue in Standard Template Library (STL)
4 #include <iostream>
5 #include <queue>
6
7 using namespace std;
8
9 void showq(queue <int> gq)
10 {
11     queue <int> g = gq;
12     while (!g.empty())
13     {
14         cout << '\t' << g.front();
15         g.pop();
16     }
17     cout << '\n';
18 }
19
20 int main()
21 {
22     queue <int> gquiz;
23     gquiz.push(10);
24     gquiz.push(20);
25     gquiz.push(30);
26
27     cout << "The queue gquiz is : ";
28     showq(gquiz);
29
30     cout << "\ngquiz.size() : " << gquiz.size();

```

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 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 linkedlist 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 // Java program to demonstrate working of Queue

```

```

1 // java program to demonstrate working of queue
2 // interface in Java
3 import java.util.LinkedList;
4 import java.util.Queue;
5
6 public class QueueExample
7 {
8     public static void main(String[] args)
9     {
10         Queue<Integer> q = new LinkedList<>();
11
12         // Adds elements {0, 1, 2, 3, 4} to queue
13         for (int i=0; i<5; i++)
14             q.add(i);
15
16         // Display contents of the queue.
17         System.out.println("Elements of queue-"+q);
18
19         // To remove the head of queue.
20         int removedele = q.remove();
21         System.out.println("removed element-" + removedele);
22
23         System.out.println(q);
24
25         // To view the head of queue
26         int head = q.peek();
27         System.out.println("head of queue-" + head);
28
29         // Rest all methods of collection interface,
30

```

Run

Output:

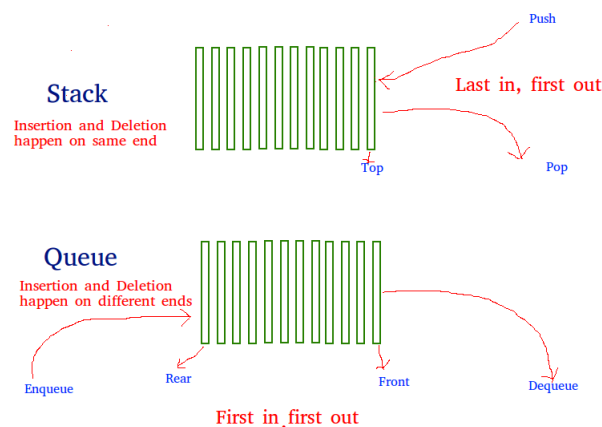
```

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

```

- 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 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 // CPP program to implement Queue using
3 // two stacks with costly enqueue()
4 #include <bits/stdc++.h>
5 using namespace std;
6
7 struct Queue {
8     stack<int> s1, s2;
9
10    void enqueue(int x)
11    {
12        // Move all elements from s1 to s2
13        while (!s1.empty()) {
14            s2.push(s1.top());
15            s1.pop();
16        }
17
18        // Push item into s1
19        s1.push(x);
20
21        // Push everything back to s1
22        while (!s2.empty()) {
23            s1.push(s2.top());
24            s2.pop();
25        }
26    }
27
28    // Dequeue an item from the queue
29    int dequeue()
30    {

```

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 // CPP program to implement Queue using
3 // two stacks with costly dequeue()
4 #include <bits/stdc++.h>
5 using namespace std;
6
7 struct Queue {
8     stack<int> s1, s2;
9
10    // Enqueue an item to the queue
11    void enqueue(int x)
12    {
13        // Push item into the first stack
14        s1.push(x);
15    }
16
17    // Dequeue an item from the queue
18    int dequeue()
19    {
20        // if both stacks are empty
21        if (s1.empty() && s2.empty()) {
22            cout << "Q is empty";
23            exit(0);
24        }
25
26        // if s2 is empty, move
27        // elements from s1
28        if (s2.empty()) {
29            while (!s1.empty()) {
30                s2.push(s1.top());

```

Run

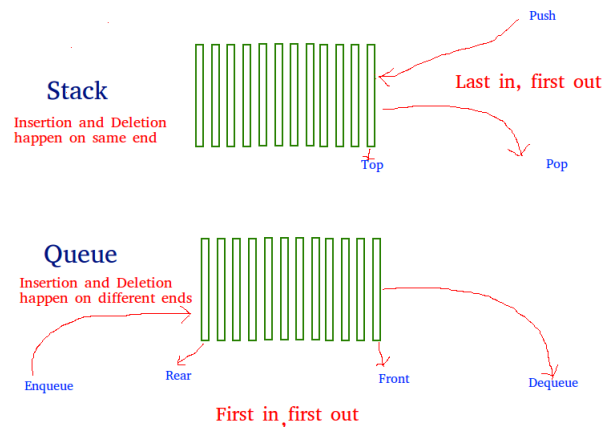
Java**Output:**

1 2 3

- 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 at 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 // C++ program to implement a stack using
3 // two queues
4 #include<bits/stdc++.h>
5
6 using namespace std;
7
8 // Stack class
9 class Stack
10 {
11     // Two inbuilt queues
12     queue<int> q1, q2;
13
14     // To maintain current number of
15     // elements
16     int curr_size;
17
18     public:
19     Stack()
20     {
21         curr_size = 0;
22     }
23
24     // Function to implement push() operation
25     void push(int x)
26     {
27         curr_size++;
28
29         // Push x first in empty q2
30         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 // Program to implement a stack using two queues
3
4 #include<bits/stdc++.h>
5 using namespace std;
6
7 // Stack class
8 class Stack
9 {
10     queue<int> q1, q2;
11     int curr_size;
12
13     public:
14     Stack()
15     {
16         curr_size = 0;
17     }
18
19     void pop()
20     {
21         if (q1.empty())
22             return;
23
24         // Leave one element in q1 and
25         // push others in q2.
26         while (q1.size() != 1)
27         {
28             q2.push(q1.front());
29             q1.pop();
30         }
```

Run

Output :

```
current size: 4
4
3
2
current size: 2
```

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 current window of k elements. An element is useful if it is in current window and is greater than all other elements on left side of it in current window. We process all array elements one by one and maintain Qi to contain useful elements of current window and these useful elements are maintained in sorted order. The element at front of the Qi is the largest and element at rear of Qi is the smallest of 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()])
}

```

[🚩 Report An Issue](#)

If you are facing any issue on this page. Please let us know.



(<https://www.geeksforgeeks.org/>)

📍 5th Floor, A-118,
Sector-136, Noida, Uttar Pradesh - 201305

✉ feedback@geeksforgeeks.org (mailto:feedback@geeksforgeeks.org)

(<https://www.facebook.com/geeksforgeeks.org/>)(https://www.instagram.com/geeks_for_geeks/)(<https://in.linkedin.com/company/geeksforgeek>

Company

About Us (<https://www.geeksforgeeks.org/about/>)

Careers (<https://www.geeksforgeeks.org/careers/>)

Privacy Policy (<https://www.geeksforgeeks.org/privacy-policy/>)

Learn

Algorithms (<https://www.geeksforgeeks.org/fundamentals-of-algorithms/>)

Data Structures (<https://www.geeksforgeeks.org/data-structures/>)

Lanquaaes (<https://www.geeksforaeks.org/category/program-output/>)

Practice	Contribute
Courses (https://practice.geeksforgeeks.org/courses/)	Write an Article (https://www.geeksforgeeks.org/contribute/)
Company-wise (https://practice.geeksforgeeks.org/company-tags/)	Write Interview Experience (https://www.geeksforgeeks.org/write-interview-experience/)
Topic-wise (https://practice.geeksforgeeks.org/topic-tags/)	Internships (https://www.geeksforgeeks.org/internship/)
How to begin? (https://practice.geeksforgeeks.org/faq.php)	Videos (https://www.geeksforgeeks.org/how-to-contribute-videos-to-geeksforgeeks/)