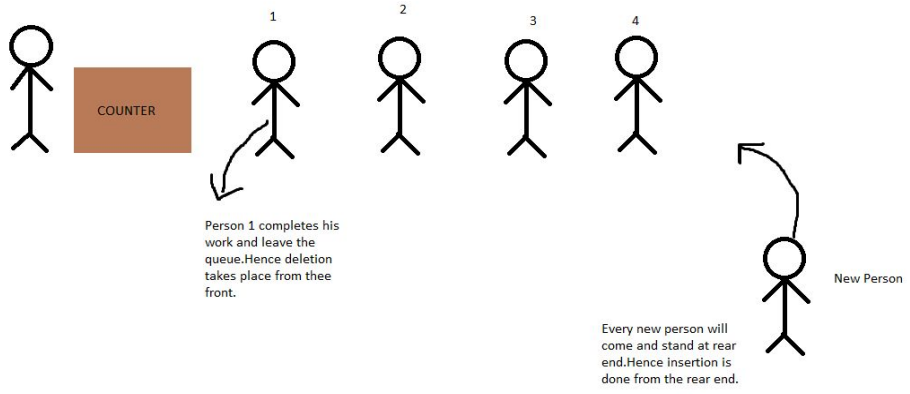
GeeksMan
Data Structure
Lesson 1
Queue Data Structure

Queue

The next Data Structure that we are going to discuss is Queue Data Structure. Just like Stack, Queue also comes under the category of logical Data Structure. The main difference between stack and queue is the principal on which these Data Structure operates. Stack works on the Principle of LIFO (Last In First Out), whereas **Queue works on the principle of FIFO (First in First out)**.

Now, let us understand the principle of FIFO (First in First out) . As the name suggests , the element which is inserted first is also deleted first . Unlike stack, where all the insertion and deletion operations were performed on the top of the stack, in Queue, the insertion and deletion operations are performed at different ends. Every new element is inserted at the rear end of the queue whereas the deletion always takes place from the front end.

Let us understand the Concept of FIFO with the help of a real life example. Suppose you are standing in a straight line in a bank to withdraw money and waiting for your turn. There are still many people ahead of you in the line and many people behind you waiting for their respective turns. Now, every new person who enters the bank to withdraw money has to stand at the rearest end of the line. Hence every new individual is at the rear end of the line. Similarly, every individual in the line can perform the transaction only and only if the person ahead of him has completed his/her transaction and has left the line. For ex:- The second person in the line can not perform transactions till the first person has not completed his work and same is the case with other people in the line. Hence deletion in the line takes place from front. This line is working on the Principle of FIFO and hence it is a Queue.



ADT(Abstract Data Type of Queue) Of Queue

Now, we know that the ADT of any Data structure consists of two parts i.e Data Representation of that Data Structure and Operation we can perform on that Data Structure. So , ADT Of Queue means what things we need to represent Queue in memory and operations we can perform on Queue . Now let's talk about both of these things on Queue in Detail:-

1.)Data Representation:-

To represent Queue in memory, we require 3 things:-

1)Space for Storing Elements :-

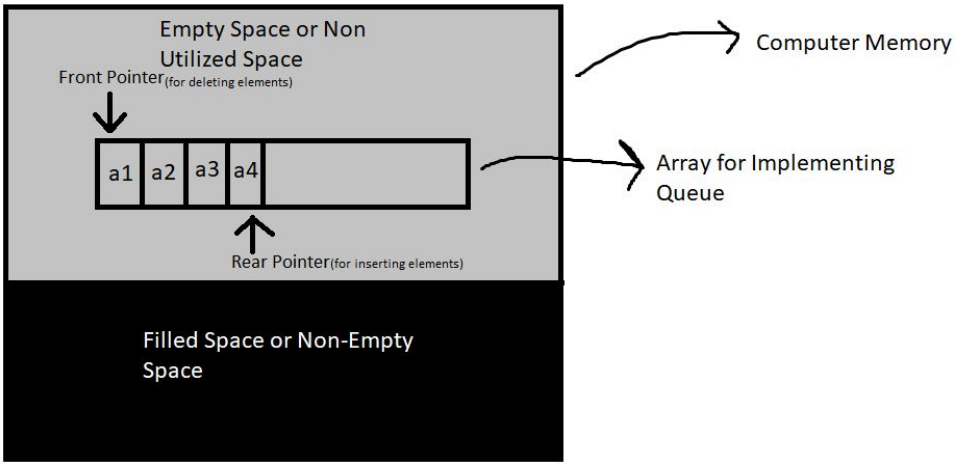
We know that Queue is a Logical Data Structure which can be implemented using any one of the Physical Data Structure i.e Array or Linked list.Now to create an array or linked list and store elements inside them,we require some empty space in memory.

2)Front Pointer:-

As already discussed , elements are deleted from the front in a queue . Hence we require a front pointer which will keep track of the current element in the front and will delete it whenever required.

3)Rear Pointer:-

We also know that new elements are inserted inside a queue from the rear End . Hence we need a rear pointer which will always point to the rear end of the queue and will insert a new element whenever needed.



Data Representation of Queue

2.)Operations on Queue:-

So Basically 6 types of Operations can be performed on Queue.

1) **enqueue(x)** Insert an element 'x' inside the queue from the rear end.

2) **dequeue()** Delete the element from the front side of the queue.

3) **isEmpty()** Finding out whether the Queue is Empty or not.

4) **isFull()** Finding out whether the Queue is Full or not.

5) **first()** Finding the current first element of the queue i.e element pointed by the front pointer.

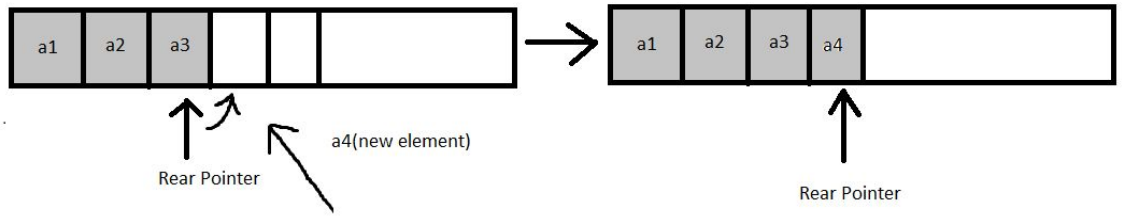
6) **last()** Finding the current last element of the queue i.e element pointed by the rear pointer.

Implementing Queue using Arrays

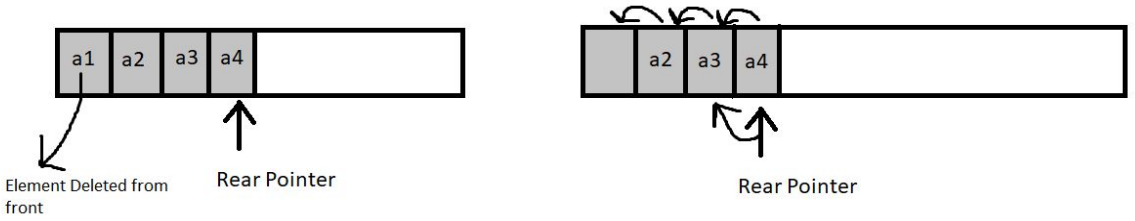
Array is a common physical data structure which can be used to implement Queue. Now, we can implement Queue using one pointer as well as by using 2 pointers. So let us Explore both of them.

A) By using Single Pointer:-

Now let us implement Queue using a single pointer only. Since elements are inserted at the rear end of the Queue , we will name our pointer **rear pointer** . Now,if we want to insert a new element inside the queue , we will just increment the value of the rear pointer and insert the element and hence , Insertion takes only **O(1)** time complexity . But the problem comes while deleting the element . Since we are not using any second pointer to keep track of the current first element , we will always delete the element at index 0 i.e a[0] . Now after deleting a[0] , that space will become empty and hence we have to shift all elements in the array by one position in the left direction to fill the empty space and again delete a[0] if required. Now if we have 'N' elements in array, then we have to shift N-1 elements in array every time whenever we delete an element . Hence deletion in array using Single pointer takes **O(N)** time complexity in average and this is the Major Drawback of Implementing Queue by just using a single pointer and hence we will move to double pointer approach.



Insertion of new Element in queue using a Single Pointer just takes O(1) time complexity



Deletion of element takes O(n) time complexity if Implementing Queue using Single Pointer since shifting of Elements has to be done

B)Implementing Queue Using Two Pointers:-

Now , since deletion using Single Pointer takes O(N) time and this is not what we want , we will change our approach a little bit to implement deletion in constant time complexity just like Insertion . We will make a front pointer which will always point to the current first Element in the Queue.

Now whenever the front most element of Queue is deleted , we will shift our front pointer to the next index of the array and hence our front pointer will be pointing to the current first element of the queue and that can be also deleted using the front pointer.

Hence , in this case we are just shifting our Single front pointer to the next index which takes **O(1) Time Complexity** rather than shifting N-1 elements which takes O(N) time complexity .

This approach for removing elements from the queue is faster , more efficient and easy to implement . Hence, we will implement a queue with an array using two separate pointers **front** and **rear** rather than using one single pointer.

Now let us define the structure for Implementing Queue using Array.

```
1.  struct queue
2.  {
3.  int *Q;//pointer pointing to array
4.  int size;//size variable to store size of array
5.  int front;//front pointer pointing to current front element in array
6.  int rear;//rear pointer pointing to the rear most element of array
7.  }
```

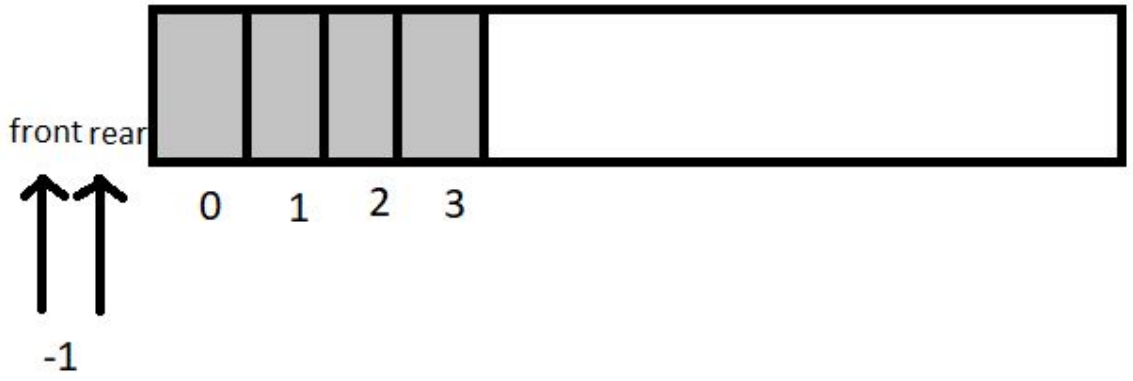
Queue Empty and Full Condition

Let us talk about Queue empty and Queue full conditions . We should know when we will say that a given queue is empty and when a given queue is full in order to prevent the violation of queue structure . We cannot delete anything from an empty queue and cannot add elements to a completely filled queue and hence knowing the Queue empty and Full Condition is important.

A)Queue Empty condition:-

A queue is said to be empty when front and rear pointer are Pointing to the same Index i.e when

Front==Rear



In the above Diagram , both front and rear are pointing to the same index and hence the queue is Empty. If we want to add a new element inside the queue , we must increment the rear pointer by 1 and add our new element. Now if we want to delete this inserted element , we will increment our front pointer by 1 and delete this element and the queue will become empty and again , front and rear will be pointing to the same index. Hence when both front and rear points to the same index , our Queue will become empty.

B)Queue full condition:-

Queue full condition is more easy to visualize and understand as compared to Queue empty condition . As we have seen , the stack is said to be full if the top pointer value becomes *equal to size of the (array - 1)*. Similarly , in a queue , since the data is inserted from the rear end of the array , a queue will be said to become full if the value of the rear pointer becomes equal to *size of (array-1).*

Hence , queue full condition is :

Rear == size-1

Now , let's Implement Queue and its operation using array and Do some Coding.

```
#include <bits/stdc++.h>
#include <iostream>
using namespace std;

//defining a structure Queue
struct Queue
{
   int size;
   int front;
   int rear;
   int *Q;
};

//create function to create a queue of a given size
void create(struct Queue *q , int size)
{
   q->size=size;
   q->front = q->rear = 0;
   q->Q = (int *)malloc(q->size*sizeof(int));
}

```

```
22. //enqueue function to enter an element from the rear end of the queue
23. void enqueue(struct Queue *q,int x)
24. {
25.   if(q->rear==q->size-1)
26.      printf("Queue is Full");
27.   else
28.   {
29.      q->rear++;
30.      q->Q[q->rear]=x;
31.   }
32. }
33.
34. //dequeue function to remove the front element from the queue
35. int dequeue(struct Queue *q)
36. {
37.   int x;
38.   if(q->front==q->rear)
39.   {
40.         printf("Queue is Empty\n");
41.      return -1;
42.   }
43.   else
44.   {
45.      q -> front++;
46.      x = q -> Q[q->front];
47.      return x;
48.   }
49. }
50.
51. //Display function to print all the elements of the queue
52. void Display(struct Queue q)
53. {
```

```
54.  int i;
55.      cout<<"The elements in the queue are: \n";
56.  for(i=q.front+1;i<=q.rear;i++)
57.  {
58.     cout<<q.Q[i];
59.         cout<<"\n";
60.  }
61. }
62. int main()
63. {
64.  //structure variable q is declared
65.  struct Queue q;
66.  //creating queue of size 5
67.  create(&q,5+1);
68.  //enqueue 10 , 20 and 30 in the queue
69.  enqueue(&q,10);
70.  enqueue(&q,20);
71.  enqueue(&q,30);
72.  //display all the elements present in the queue
73.  Display(q);
74.  //removing front element from the queue and print it
75.      cout<<"The removed element is : "<<dequeue(&q)<<"\n";
76.
77.  return 0;
78. }
```

Link to above code:- https://sapphireengine.com/@/grhj00

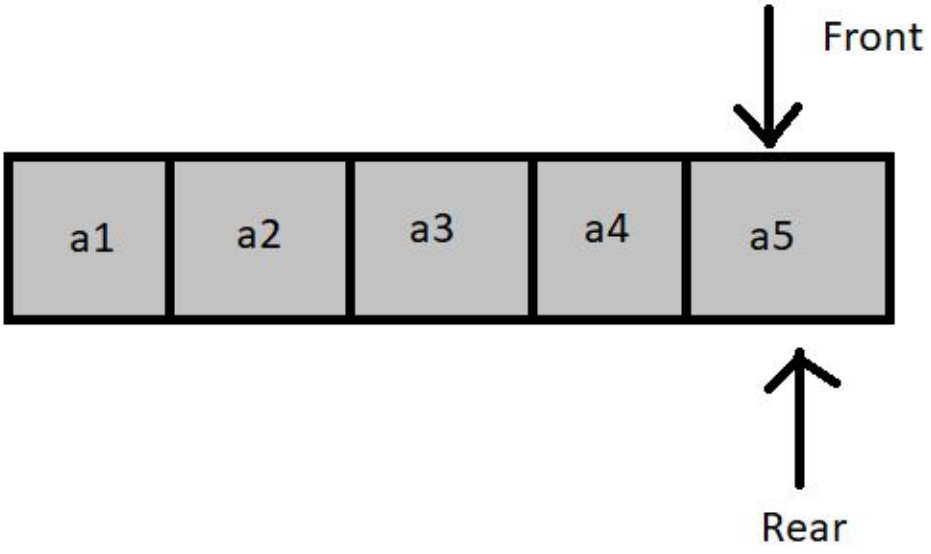
Disadvantage of Implementing Queue using Array

1)Waste of Space:

While deleting elements , we are moving the front pointer to the next index of the array . For example after deleting the first element of array i.e index 0 , our front pointer will move to the next index i.e index 1 .Now we will never go back to that index of array since there is no element present there and hence that index is useless for us . But still , since that index is part of our array it will occupy some space in our memory which we are never going to use . Similar thing will happen if we remove one more element and hence more space will get unused and hence more space will get wasted . Hence there is a waste of space while implementing a queue using arrays.

2)A condition will arise when a queue will be Empty as well as full

Now let us see this condition with the help of a diagram.



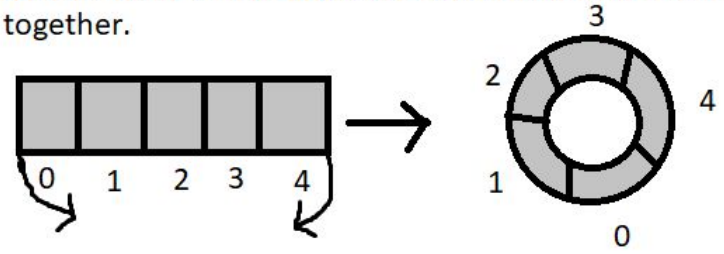
Now in the above case , the queue is empty as well as full and this is very ambiguous . The rear pointer is pointing to the last index of the array and hence our queue is full . On the other hand , as all the elements are deleted from the queue and hence now the front is pointing to the same index as that of the rear . Since , rear==front , the queue is empty as well . But this is impossible . A queue can't be full and empty at the same time . The problem is arising because we are only using each location only once . In ideal cases , as soon as an element is removed from the queue and that index becomes empty , we should use it again to store any new incoming element . But we are not doing any such thing here , we are just using each location once and hence this problem arises .

Solution of these Disadvantages

CIRCULAR QUEUE

Now ,the biggest problem we are facing while using an array for implementing a queue is that we are not reusing the location from where the element is deleted . As soon as the element is deleted from a particular index , we are abandoning that index and not using that index anymore . But in ideal cases , we should reuse the index to store any new incoming element. This gives rise to the concept of circular queue. In a circular queue we cover up the limitation of a linear queue of not using the same location twice by just adding one single condition . As soon as the queue becomes full (i.e rear pointer value becomes equal to size-1) , instead of terminating the program and saying the queue is full , we redirect our rear pointer to the beginning of the array (i.e index 0) by using modulus operator(%) . Now , if the element has been deleted from that index , we can insert our new element there by using the rear pointer as we normally do , hence using the same location twice , we cover up the limitation of the linear queue . Hence here we are avoiding wastage of space in the ambiguous case by just adding one condition.

Note:-Circular queue is called circular queue just because of its behaviour of circularly inserting new elements when the queue becomes Full.In memory, it is represented in linear form only just like any other array and not Circularly.But as we visualized stack as an array placed vertically,for simplicity and more clearity,you can visualize circular queue as an circular array made by joining two extreme ends of array together.




```
#include <bits/stdc++.h>
#include <iostream>
using namespace std;
struct Queue
{
    int size;
    int front;
    int rear;
    int *Q;
};
void create(struct Queue *q,int size)
{
    q->size=size;
    q->front=q->rear=0;
    q->Q=(int *)malloc(q->size*sizeof(int));
}
void enqueue(struct Queue *q,int x)
{
    if((q->rear+1)%q->size==q->front)
        {
            cout<<"Queue is full\n";
```

```
        }
    else
        {
            q->rear=(q->rear+1)%q->size;
            q->Q[q->rear]=x;
        }
}
int dequeue(struct Queue *q)
{
    int x=-1;

    if(q->front==q->rear)
        cout<<"Queue is Empty\n";
    else
    {
        q->front=(q->front+1)%q->size;
        x=q->Q[q->front];
    }
    return x;
}
void Display(struct Queue q)
{
    int i=q.front+1;
    cout<<"Elements in circular queue are"<<"\n";
    do
        {
            cout<<q.Q[i]<<"\n";
            i=(i+1)%q.size;
        }
       while(i!=(q.rear+1)%q.size);
}

int main()
{
struct Queue q;
int k;
cout<<"Enter the size of circular queue"<<"\n";
cin>>k;
create(&q,k+1);
```

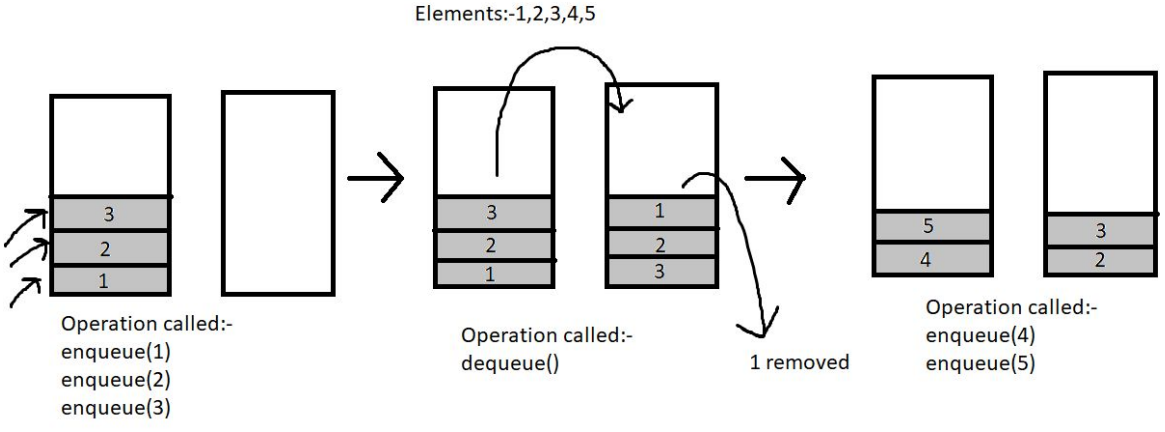
```
61.
62.  enqueue(&q,10);
63.  enqueue(&q,20);
64.  enqueue(&q,30);
65.  enqueue(&q,40);
66.  enqueue(&q,50);
67.  enqueue(&q,60);
68.
69.  Display(q);
70.  cout<<"Element deleted from circular queue is"<<"\n";
71.
72.  cout<<dequeue(&q)<<"\n";
73.  return 0;
74.  }
```

Link of code:- https://sapphireengine.com/@/3w5id0

IMPLEMENTING QUEUE USING 2 STACKS

Now we have Seen 2 logical data structures i.e Stacks and Queue . Now here comes an interesting problem . Can we Implement queue using 2 stacks i.e perform enqueue and dequeue operation on a given set of elements using 2 stacks . Now , we will discuss the logic required to solve this problem and writing the code will be you .

Logic:-The logic for this question is very simple . We have 2 stacks and we have to do two operations i.e enqueue and dequeue . Now we will enqueue every new element in stack 1 and when a call to dequeue will be made to remove an element , we will check whether the second stack is empty or not . If it is empty , then we will transfer all elements from stack 1 to stack 2 making stack 1 empty . By doing this , the element to be removed from the queue will acquire the top most position in stack 2 and hence we will remove that element . Now, if a call to dequeue is made again and stack 2 is non empty we will remove the topmost element from stack 2 as while shifting elements from stack 1 to stack 2 , we have reversed the order of elements and hence the elements in stack 2 if viewed from top to bottom are in the same order as if elements viewed from front to rear in queue and the top most element in stack 2 will always contain the element which we want to dequeue from queue based on the principle of FIFO.



Queue in Standard Template Library (STL)

Queues are a type of container adaptors which operate in a first in first out (FIFO) type of arrangement . Till now , we were applying the concept of queue using arrays and structures . In this section , we will study about Queue in STL.
Following are the functions supported by queue in STL :

1.	empty()	:	Returns whether the queue is empty.
2.	size()	:	Returns the size of the queue.
3.	front()	:	returns a reference to the first element of the queue
4.	back()	:	returns a reference to the last element of the queue
5.	push(g)	:	adds the element 'g' at the end of the queue.
6.	pop()	:	deletes the first element of the queue.

Example :

```
1.  #include<bits/stdc++.h>
2.  using namespace std;
3.  int main()
4.  {
5.      queue<int>qq;
6.      qq.push(1);
7.      qq.push(2);
8.      qq.push(3);
9.
10.     cout << "The queue is : ";
11.
12.     queue <int> g = qq;
13.     while (!g.empty())
14.     {
```

```
15.             cout  << g.front() << " ";
16.             g.pop();
17.         }
18.
19.         cout << "\nsize of queue =  : " << qq.size();
20.         cout << "\nfront element =  : " << qq.front();
21.         cout << "\nback element = : " << qq.back();
22.
23.         cout << "\nAfter Popping and element  :" ;
24.         qq.pop();
25.         cout << "\nsize of queue =  : " << qq.size();
26.         cout << "\nfront element =  : " << qq.front();
27.         cout << "\nback element = : " << qq.back();
28.         return 0;
29.     }
```

Output :

stdout

```
The queue is : 1 2 3
size of queue =  : 3
front element =  : 1
back element = : 3
After Popping and element  :
size of queue =  : 2
front element =  : 2
back element = : 3
```

Link of the code :
https://sapphireengine.com/@/vt1d69

Questions for practice:

- **Queue push and pop**

https://practice.geeksforgeeks.org/problems/queue-designer/1

- **c++stl(set5)**

https://practice.geeksforgeeks.org/problems/c-stl-set-5-queue/1

- **Implement queue using array**

https://practice.geeksforgeeks.org/problems/implement-queue-using-array/1

- **Queue Reversal**

https://practice.geeksforgeeks.org/problems/queue-reversal/1/?category[]=Stack&category[]=Queue&page=1&query=category[]Stackcategory[]Queuepage1

- **Queue using two stack**

https://practice.geeksforgeeks.org/problems/queue-using-two-stacks/1

- **Stack-using-two-queues**

https://practice.geeksforgeeks.org/problems/stack-using-two-queues/1