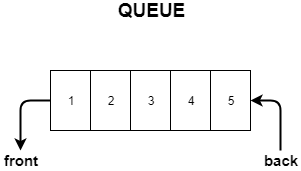
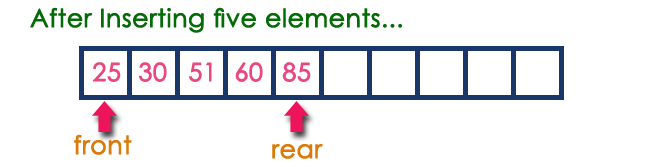
Queue

**Queue**is a linear data structure following the FIFO(First In, First Out) principle.

/rear

# Example

Queue after inserting 25, 30, 51, 60 and 85.



# Operations on a Queue

The following operations are performed on a queue data structure...

1. **enQueue(value) - (To insert an element into the queue)**
2. **deQueue() - (To delete an element from the queue)**
3. **display() - (To display the elements of the queue)**

Queue data structure can be implemented in two ways. They are as follows...

1. **Using Array**
2. **Using Linked List**

**Enqueue Operation**

* Enqueue means inserting an element in the queue.A new element in a queue is inserted at the back of the queue.

Check if the **queue is full**.

If its full

    we can't insert data.

Otherwise

     insert the data in **arr[rear]** and **increment the rear variable by 1.**

.

**Dequeue Operation**

Dequeue means removing an element from the queue. Since queue follows the FIFO principle we need to remove the element of the queue which was inserted at first. Naturally, the element inserted first will be at the front of the queue so we will remove the **front** element and let the element behind it be the new front element.

Check if the queue is **empty**

if it's **empty**

    We can't dequeue an element from the empty queue.

Otherwise

    Print/return **arr[front]** and **increment the front by 1**.

# display() - Displays the elements of a Queue

We can use the following steps to display the elements of a queue...

* **Step 1 -**Check whether **queue** is **EMPTY**. (**front == rear**)
* **Step 2 -**If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
* **Step 3 -**If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front+1**'.
* **Step 4 -**Display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i**' value reaches to **rear** (**i** <= **rear**)

#### isQueueFull()

This function will return **true(1)** if the queue is full.

Otherwise, it will return **false(0)**.

**int** **isQueueFull**();

#### isQueueEmpty()

This function will return **true(1)** if the queue is empty.

Otherwise, it will return **false(0)**.

**int** **isQueueEmpty**();

## Simple Queue Application

In a bank, people are standing in the queue. The person who stands first in the line will be served first.

Similarly, the person who stands last in the line will be served at the end.

This is the simplest example of First In First Out (FIFO) principle.

## Queue data structure using array

## Write all the above operations using arrays.

**Queue using linked list**

**Queue using an array - drawback**

If we implement the queue using an array, we need to specify the array size at the beginning(at compile time).We can't change the size of an array at runtime. So, the queue will only work for a fixed number of elements.

**Solution:**We can implement the queue data structure using the linked list.In the linked list, we can change its size at runtime.

#### Node structure for Queue

**struct** node

{

**int** data;

**struct** node \*next;

};

#### To point the front and rear node

**struct** node \*front = NULL, \*rear = NULL;

#### Enqueue function

Enqueue function will add the element at the end of the linked list.

Using the rear pointer, we can track the last inserted element.

1. Declare a new node and allocate memory for it.

2. If front == NULL,

     make both front and rear points to the new node.

3. Otherwise,

     add the new node in rear->next.

     make the new node as the rear node. i.e. rear = new node

void enqueue(int val)

{

struct node \*newNode = malloc(sizeof(struct node));

newNode->data = val;

newNode->next = NULL;

//if it is the first node

if(front == NULL && rear == NULL)

//make both front and rear points to the new node

front = rear = newNode;

else

{

//add newnode in rear->next

rear->next = newNode;

//make the new node as the rear node

rear = newNode;

}

}

#### Dequeue function

Dequeue function will remove the first element from the queue.

1.Check whether the queue is empty or not

2.If it is the empty queue (front == NULL)

     We can't dequeue the element.

3.Otherwise,

     Make the front node points to the next node. i.e front = front->next;

     if front pointer becomes NULL, set the rear pointer also NULL.

     Free the front node's memory.

**void** **dequeue**()

{

//used to freeing the first node after dequeue

**struct** node \*temp;

**if**(front == NULL)

printf("Queue is Empty. Unable to perform dequeue**\n**");

**else**

{

//take backup

temp = front;

//make the front node points to the next node

//logically removing the front element

front = front->next;

//if front == NULL, set rear = NULL

**if**(front == NULL)

rear = NULL;

//free the first node

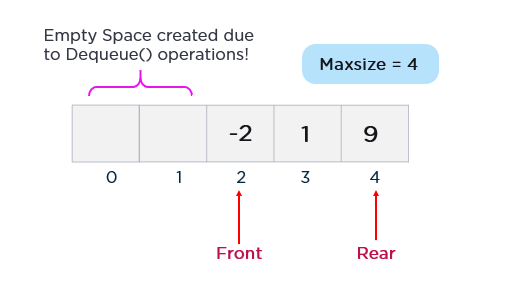
free(temp);

}

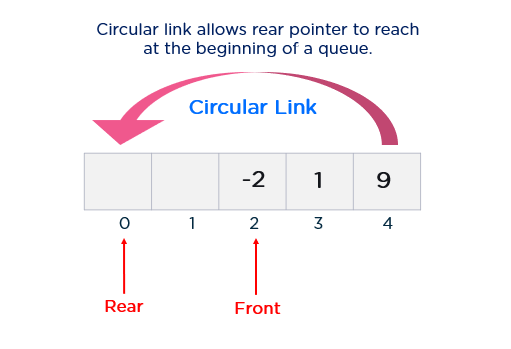
}

# Circular Queue

## Why Circular Queue ?

Implementation of a linear queue brings the drawback of memory wastage. In the case of a linear queue, when the rear pointer reaches the MaxSize of a queue, there might be a possibility that after a certain number of dequeue() operations, it will create an empty space at the start of a queue. 

Additionally, this newly created empty space can never be re-utilized as the rear pointer reaches the end of a queue. Hence, experts introduced the concept of the circular queue to overcome this limitation.



As shown in the figure above, the rear pointer arrives at the beginning of a queue with the help of a circular link to re-utilize the empty space to insert a new element. This simple addition of a circular link resolves the problem of memory wastage in the case of queue implementation.

What is Circular Queue in a Data Structure?

A circular queue is an extended version of a linear queue as it follows the First In First Out principle with the exception that it connects the last node of a queue to its first by forming a circular link. Hence, it is also called a Ring Buffer.

## Circular_queue_representation Representation of Circular Queue using Arrays

The process of array implementation starts with the declaration of array and pointer variables. You use the following statements for it:

#define MAX\_SIZE 5 //global value assignment to max\_size variable

int a[MAX\_SIZE];

int front = -1;

int rear = -1;

After this, you will begin with the implementation of circular queue operations.

1. The first primary queue operation that allows you to manage data flow in a queue is Enqueue(). For implementing this function, you have to check if the queue is empty. If a circular queue is empty, you must set both the front and rear pointer to 0 manually. This condition can be implemented as follows:

void enqueue(int x)

{

if (front == -1 && rear == -1)

{

front = rear = 0;

}

else if ((rear + 1) % MAX\_SIZE == front)

{

printf("queue is full\n");

return;

}

else

rear = (rear + 1) % MAX\_SIZE;

a[rear] = x;

}

2. The next operation is Dequeue(). The Dequeue() operation removes the element from the front node of a queue. Further, an element can only be deleted when there is at least an element to delete, i.e., Rear > 0 (queue shouldn’t be empty). If the queue is empty, then you must write an underflow error and exit.

void dequeue()

{

if (front == -1 && rear == -1)

{

printf("queue is empty \n");

return;

}

else if (front == rear)

{

front = rear = -1;

}

else

front = (front + 1) % MAX\_SIZE;

}

## Implementation of a Circular Queue Using a Linked List

The [linked list](https://www.simplilearn.com/tutorials/data-structure-tutorial/linked-list-in-data-structure) implementation of a circular queue begins with the declaration of a node and pointer variables. For the creation of a node, you will use structure in C as follows:

|  |
| --- |
| // Declaration of node using structure in C programming    struct node    {  int data;  struct node \*next;    };    struct node \*front=-1;    struct node \*rear=-1; |

Implement circular queue operation now.

1. First, you must implement Enqueue() operation. In this function, you will allocate the memory for a new node using the following snippet of code:

|  |
| --- |
| void enqueue(int x)    {  struct node \*newnode;  newnode=(struct node \*)malloc(sizeof(struct node));  //malloc function helps in allocating memory |

After creating a new node, you must also insert values into both data and reference fields. Data will be an argument provided by the user, and the reference field will be set to null initially.

|  |
| --- |
| newnode->data=x;       newnode->next=0; |

There are two conditions for inserting a new node into the linked circular queue.

In the first condition, you will insert a new node into an empty queue. In this case, both the front and rear pointer must be Null. You must check it using the condition rear == -1. If it is true, the new element will be added as the first element of the queue, and both the front and the rear pointer will also be updated to point to this new node.

|  |
| --- |
| if(rear == -1){    front = rear = newnode;    Rear->next = front      } |

In the second case, the queue already contains over one data element. Now, the condition rear == -1 becomes false. In this scenario, you will just update the rear pointer to point to the new node. You should also change the address field of the previous node to establish a new link.

|  |
| --- |
| else  {        rear->next=newnode;        rear=newnode;        rear->next=front;   } |

2. The dequeue() operation is accountable for deleting the data element from the front node of a queue. While implementing this operation, you will first check if the queue is empty as you cannot delete the element if there is no element in a queue.

|  |
| --- |
| void dequeue()     {     struct node \*temp;   // declaration of pointer node     temp=front;     if((front==-1)&&(rear==-1))      {          printf("\nQueue is empty");      } |
| else if(front==rear)     {         front=rear=-1;         free(temp);     } |
| else     {         front=front->next;         rear->next=front;         free(temp);     } |

Applications of a Circular Queue

* CPU Scheduling: In the Round-Robin Scheduling Algorithm, a circular queue is utilized to maintain processes that are in a ready state.
* Traffic System: Circular queue is also utilized in traffic systems that are controlled by computers. Each traffic light turns ON in a circular incrementation pattern in a constant interval of time.