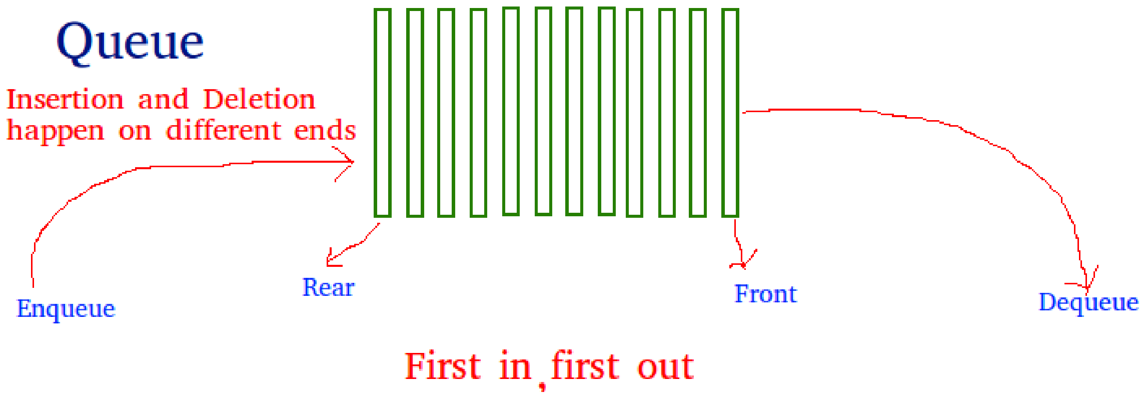
QUEUE DATA STRUCTURE

- Queue is a linear data structure, somewhat similar to Stacks.
- Unlike stacks, a queue is open at both its ends.
- One end is always used to insert data (enqueue) and the other is used to remove data (dequeue).
- Queue follows First-In-First-Out methodology (FIFO), i.e., the data item stored first will be accessed first.
- A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first.

- More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

- The following diagram given below tries to explain queue representation as data structure −



- As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures.

- For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

Basic operations associated with queues are

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.

Enqueue Operation

- Queues maintain two data pointers, front and rear.
- Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue –

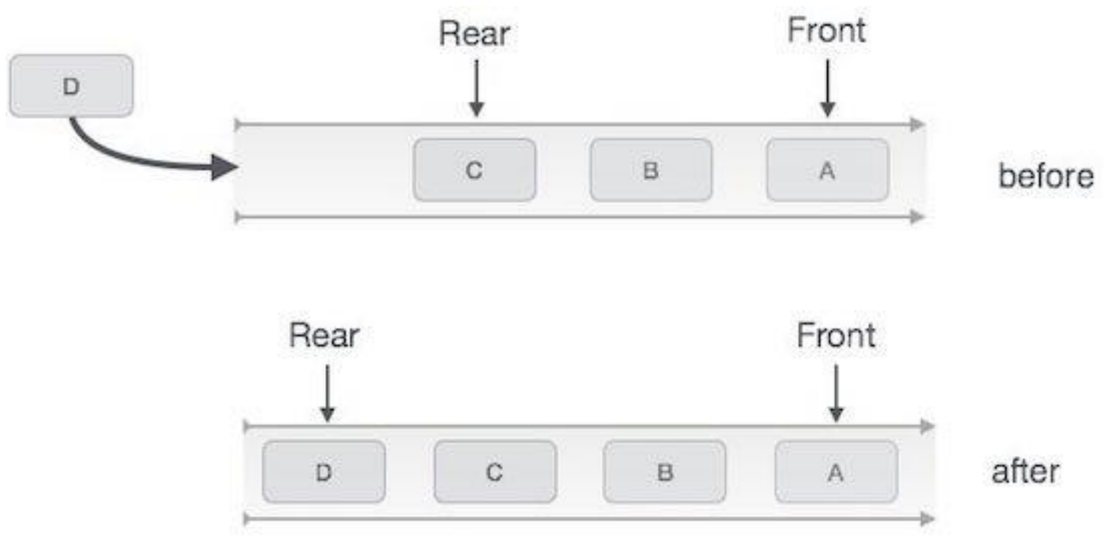
Step 1 − Check if the queue is full.

Step 2 − If the queue is full, produce overflow error and exit.

Step 3 − If the queue is not full, increment rear pointer to point the next empty space.

Step 4 − Add data element to the queue location, where the rear is pointing.

Step 5 − return success.



Queue Enqueue

Algorithm for enqueue operation

```
procedure enqueue(data)

   if queue is full

      return overflow

   endif

   rear ← rear + 1

   queue[rear] ← data

   return true

end procedure
```

Dequeue Operation

- Accessing data from the queue is a process of two tasks

  I. Access the data where front is pointing

  II. Removes the data after access.

- The following steps are taken to perform dequeue operation

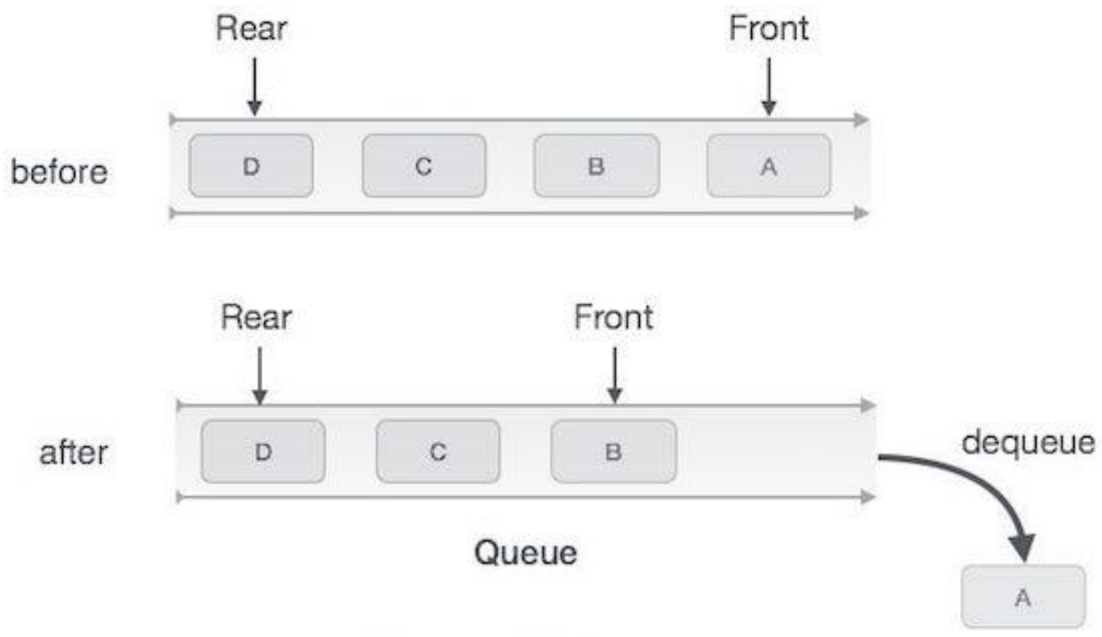
Step 1 − Check if the queue is empty.

Step 2 − If the queue is empty, produce underflow error and exit.

Step 3 − If the queue is not empty, access the data where front is pointing.

Step 4 − Increment front pointer to point to the next available data element.

Step 5 − Return success.



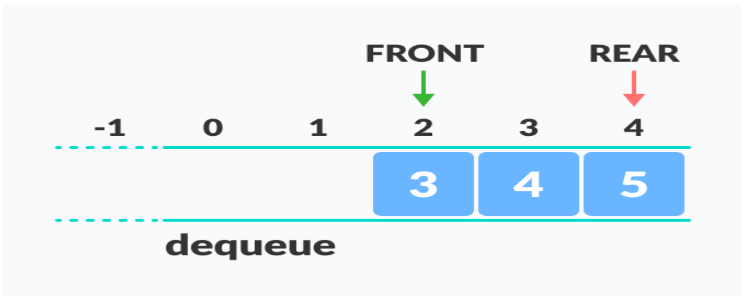
Queue Dequeue

Algorithm for dequeue operation

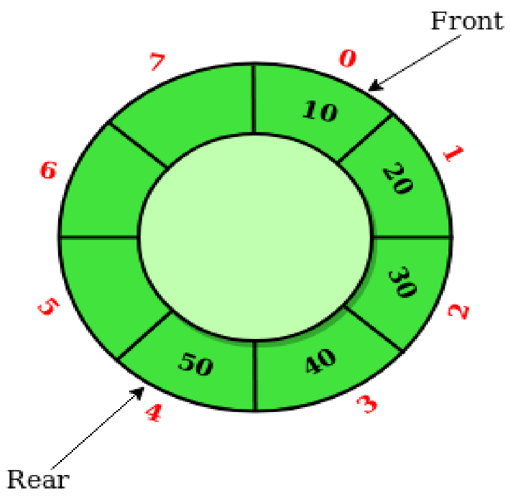
```
procedure dequeue()

   if queue is empty

      return underflow

   endif

   data←queue[front]

   front←front+1

   return data

end procedure
```

CIRCULAR QUEUE

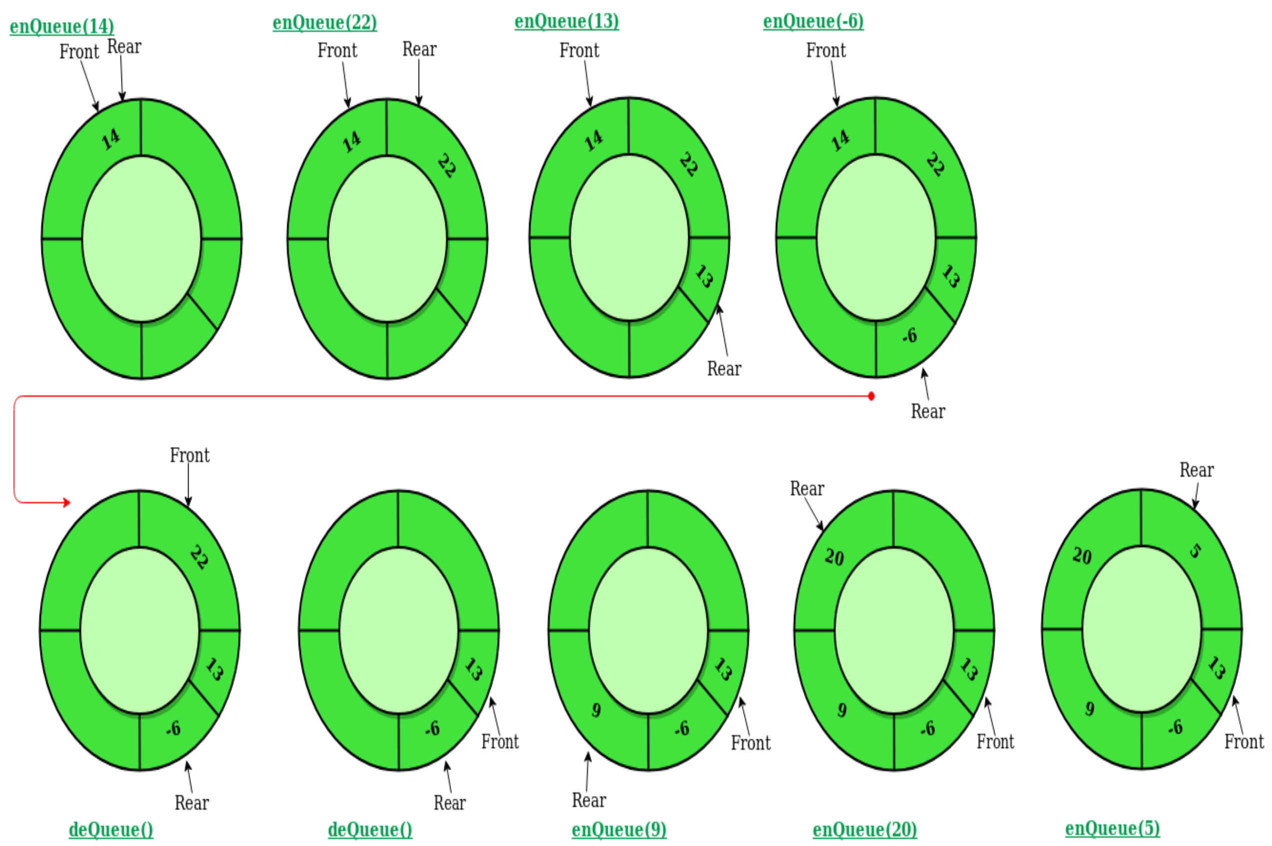
- Circular queue avoids the wastage of space in a regular queue implementation using arrays.



- As you can see in the above image, after a bit of enqueuing and dequeuing, the size of the queue has been reduced.
- Circular Queue works by the process of circular increment i.e. when we try to increment the pointer and we reach the end of the queue, we start from the beginning of the queue.
- Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called 'Ring Buffer'.



- In a normal Queue, we can insert elements until queue becomes full.
- But once queue becomes full, we cannot insert the next element even if there is a space in front of queue.



Here, the circular increment is performed by modulo division with the queue size. That is,

```
if REAR + 1 == 5 (overflow!), REAR = (REAR + 1)%5 = 0
(start of queue)
```

However, the check for full queue has a new additional case:

- Case 1: `FRONT` = 0 && `REAR == SIZE - 1`
- Case 2: `FRONT = REAR + 1`

Operations on Circular Queue

Front: Get the front item from queue.

Rear: Get the last item from queue.

enQueue(value)

- This function is used to insert an element into the circular queue.
- In a circular queue, the new element is always inserted at Rear position.

Steps:

- Check whether queue is Full
  – Check ((rear == SIZE-1 && front == 0) || (rear == front-1)).
- If it is full then display Queue is full.
- If queue is not full then, check if (rear == SIZE – 1 && front != 0) if it is true then set rear=0 and insert element.

- ### deQueue()
  - This function is used to delete an element from the circular queue.
  - In a circular queue, the element is always deleted from front position.

Steps:

- Check whether queue is Empty means check (front==-1).
- If it is empty then display Queue is empty. If queue is not empty then step 3
- Check if (front==rear) if it is true then set front=rear= -1 else check if (front==size-1), if it is true then set front=0 and return the element.

Algorithm to insert an element in circular queue

```
Step 1: IF (REAR+1)%MAX = FRONT
            Write " OVERFLOW "
            Goto step 4
        [End OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE IF REAR = MAX - 1 and FRONT ! = 0
            SET REAR = 0
        ELSE
            SET REAR = (REAR + 1) % MAX
        [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
```

Algorithm to delete an element from a circular queue

```
Step 1: IF FRONT = -1
            Write " UNDERFLOW "
            Goto Step 4
        [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
            SET FRONT = REAR = -1
        ELSE
            IF FRONT = MAX -1
                SET FRONT = 0
            ELSE
                SET FRONT = FRONT + 1
            [END of IF]
        [END OF IF]
Step 4: EXIT
```

QUEUE OPERATIONS(NORMAL QUEUE)

```
#define SIZE 5

int items[SIZE];
int front = -1, rear = -1;
```

INTIALIZATION

```
void enQueue(int value)
 {
  if (rear == SIZE - 1)
    printf("\nQueue is Full!!");
  else
{
    if (front == -1)
      front = 0;
    rear++;
    items[rear] = value;
    printf("\nInserted -> %d", value);
  }
}

void deQueue()
{
  if (front == -1)
    printf("\nQueue is Empty!!");
  else
{
    printf("\nDeleted : %d", items[front]);
    front++;
    if (front > rear)
      front = rear = -1;
  }
}
```

CIRCULAR QUEUE OPERATIONS

```
int isFull()
{
  if ((front == rear + 1) || (front == 0 &&
rear == SIZE - 1)) return 1;
  return 0;
}

int isEmpty() {
  if (front == -1) return 1;
  return 0;
}

void enQueue(int element)
 {
  if (isFull())
    printf("\n Queue is full!! \n");
  else
{
    if (front == -1) front = 0;
    rear = (rear + 1) % SIZE;
    items[rear] = element;
    printf("\n Inserted -> %d", element);
  }
}
int deQueue()
{
  int element;
  if (isEmpty())
{
    printf("\n Queue is empty !! \n");
    return (-1);
}
else
{
    element = items[front];
    if (front == rear)
   {
      front = -1;
      rear = -1;
    }
    // Q has only one element, so we reset the
    // queue after dequeing it. ?
    else
    {
      front = (front + 1) % SIZE;
    }
  printf("\n Deleted element -> %d \n", element);
   return (element);
  }
}
```