QUEUE ADT

Queue Model

A **Queue** is a linear data structure which follows **First In First Out (FIFO)** principle, in which **insertion** is performed at **rear** end and **deletion** is performed at **front** end.

Example: Waiting Line in Reservation Counter, queue in a cinema ticket counter, Scheduling of jobs in computer

OPERATIONS ON QUEUE

The fundamental operations performed on queue are

1. Enqueue
2. Dequeue

ENQUEUE:

- The process of inserting an element in the queue.

DEQUEUE:

- The process of deleting an element from the queue.

EXCEPTION CONDITIONS

Overflow: Attempt to insert an element, when the queue is full is said to be overflow condition.
Underflow: Attempt to delete an element from the queue, when the queue is empty is said to be underflow.

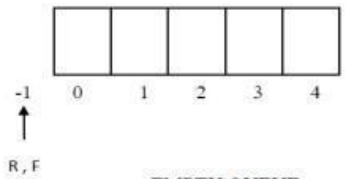
IMPLEMENTATION OF QUEUE

Queue can be implemented using array and linked list.

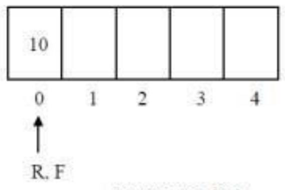
ARRAY IMPLEMENTATION

- In array implementation queue Q is associated with two pointers namely Rear and Front.
- For an empty queue Front = -1 and Rear = -1

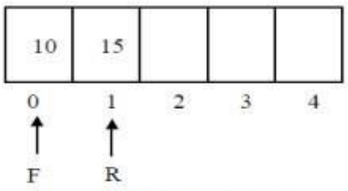
- To insert an element X onto the queue Q, the Rear is incremented by 1 and then set Q [Rear] = X
- To delete an element, the Q [front] is returned and the Front is incremented by 1.



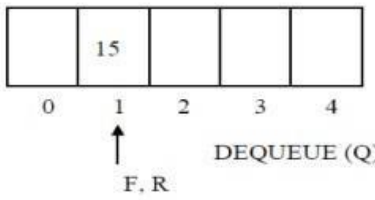
EMPTY QUEUE



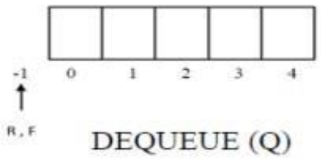
ENQUEUE (10)



ENQUEUE (15)



DEQUEUE (Q)



DEQUEUE (Q)

initialize the value of front and rear:

front = -1;

rear = -1;

1. Enqueue Operation

The process of inserting **an element at the rear end** of the queue.

Steps:

- First, the function checks whether the queue is **full**.
- If rear == MAX - 1, no more elements can be inserted, so **Queue Overflow** occurs.
- If the queue is not full:
  - When inserting the **first element**, front is set to 0.

    - The rear index is incremented.
    - The new element x is stored at q[rear].
- A message is displayed to show that the element is successfully inserted.

C++ code for Enqueue( )

```
void Enqueue(int x)
{
    if (rear == MAX - 1)
   {
        cout << "Queue Overflow\n";
    }
    else
    {
        if (front == -1)
            front = 0;
        q[++rear] = x;
        cout << x << " inserted into queue\n";
    }
}
```

2. Dequeue Operation

The process of deleting **an element from the front end** of the queue.

Steps:

- The function first checks whether the queue is **empty**.
- If front == -1 or front > rear, the queue has no elements, so **Queue Underflow** occurs.
- If the queue is not empty:
    - The element at q[front] is removed.
    - The front index is incremented to point to the next element.
- A message is printed to show the deleted element.

Note : Dequeue always removes an element from the **front** of the queue.

C++ code for Dequeue( ):

```
void dequeue()
{
    if (front == -1 || front > rear)
   {
        cout << "Queue Underflow\n";
    }
    else
    {
        cout << q[front++] << " deleted from queue\n";
    }
}
```

3. Display Operation

To **display all elements** present in the queue.

Steps:

- First, the function checks whether the queue is **empty**.
- If the queue is empty, it prints "Queue is empty".
- If the queue contains elements:
    - A loop starts from front index to rear index.
    - Each element is printed in order.
- This shows the queue contents from **front to rear**.

C++ code for Display( ):

```
void display()
 {
    if (front == -1 || front > rear)
    {
        cout << "Queue is empty\n";
    }
     else
    {
        cout << "Queue elements:\n";
        for (int i = front; i <= rear; i++)
            cout << q[i] << " ";
        cout << endl;
    }
}
```

Advantages

1. **Easy to implement** and understand.
2. **Fast processing** because memory is contiguous.
3. **No extra memory** required for pointers.
4. Suitable for **small and fixed-size queues**.

Disadvantages

1. **Fixed size** – cannot increase size during execution.
2. **Queue overflow** occurs when the array becomes full.
3. **Memory wastage** if the queue is not fully utilized.
4. Not suitable when data size is unpredictable.

Applications of Queue Using Array

1. **CPU scheduling** (ready queue with fixed size).
2. **Printer spooling**.
3. **Keyboard buffer**.
4. **Simple task scheduling**.

5. **Static data processing systems**.

LINKED LIST IMPLEMENTATION OF QUEUE

- Enqueue operation is performed at the end of the list (Rear end).
- Dequeue operation is performed at the front of the list (Front end)

Declaration of Queue using Linked List

To implement a queue with a linked list, we maintain:

A Node structure/class that contains:

- data → to store the element.
- next → pointer/reference to the next node in the queue.

Two pointers/references:

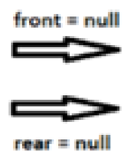
- front → points to the first node (head of the queue).
- rear → points to the last node (tail of the queue).

1. Insert Operation (Enqueue)

Enqueue means to insert a new element at the **rear end** of the queue.

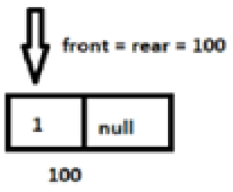
Algorithm / Steps:

1. Create a new node.
2. Store the given value x in the data field of the new node.
3. Set the next pointer of the new node to NULL.
4. Check if the queue is empty (rear == NULL):
   - If **empty**, assign both front and rear to the new node.
5. Otherwise:
   - Link the current rear node to the new node (rear->next = new node).
   - Move rear to point to the new node.
6. Display a message indicating successful insertion.



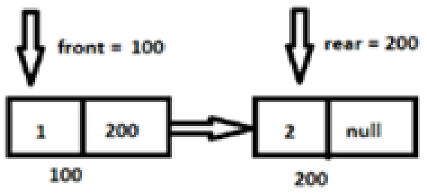
Initially front and rear are null

(A)



Enqueue 1, both front and rear point to same address 100

(B)



Enqueue 2 at the rear end, rear pointer points to the newly added element. Front still points to the first element in the queue.

(C)

C++ code for Enqueue( ):

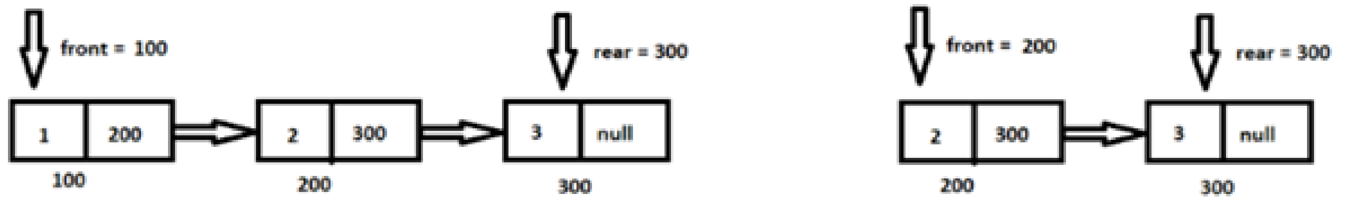
```
void insert(int x)
   {
       Node *temp = new Node;
       temp->data = x;
       temp->next = NULL;
       if (rear == NULL)
       {
           front = rear = temp;
       }
       else
       {
           rear->next = temp;
           rear = temp;
       }
       cout << x << " inserted into queue\n";
   }
```

2. Delete Operation (Dequeue)

Dequeue is the operation of **removing (deleting) an element from a Queue**.

Algorithm / Steps:

1. Check if the queue is empty (front == NULL)
   - If **empty**, display **Queue Underflow**.
2. Otherwise:
   - Store the front node in a temporary pointer.
   - Display the data of the node being deleted.
   - Move front to the next node (front = front->next).
   - Free the memory of the deleted node.
3. If after deletion front becomes NULL:
   - Set rear also to NULL (queue becomes empty).



Enqueue 3 at the rear end, rear points to the newly added element 3. Front still points to the first element of the queue.

Dequeue an element from the front. Front now points to element with data 2 and rear still points to the last element of the queue.

(A) Queue implementation using linked list (B)

C++ code for Dequeue( ):

```
void del()
{
    if (front == NULL)
    {
        cout << "Queue Underflow\n";
    }
```

```
        else
        {
            Node *temp = front;
            cout << temp->data << " deleted from queue\n";
            front = front->next;
            delete temp;
            if (front == NULL)
            {
                rear = NULL;
            }
        }
    }
```

3. Display Operation

To display all elements present in the queue from **front to rear**.

Algorithm / Steps:

1. Check if the queue is empty (front == NULL):
   - If **empty**, display “Queue is empty”.
2. Otherwise:
   - Initialize a pointer ptr to front.
   - Traverse the queue until ptr becomes NULL:
     - Print the data of each node.
     - Move ptr to the next node.
3. End traversal after reaching the last node.

C++ code for display():

```
void display()
{
    if (front == NULL)
    {
        cout << "Queue is empty\n";
    }
    else
    {
        Node *ptr = front;
        cout << "Queue elements: ";
        while (ptr != NULL)
        {
            cout << ptr->data << " ";
            ptr = ptr->next;
        }
        cout << endl;
    }
}
```

Advantages

1. **Size is flexible** (dynamic).
2. **No overflow** until memory is full.
3. **No memory wastage**.
4. **Efficient insertion and deletion**.
5. Suitable for **large and unpredictable data**.

Disadvantages

1. Uses **extra memory** for pointer.
2. **Little complex** compared to array.
3. **Slower** because of pointer traversal.
4. Careless pointer handling may cause errors.

Applications of Queue Using Linked List

1. Operating systems **(process scheduling).**
2. Network packet handling**.**
3. Message queues**.**
4. Real-time data processing**.**
5. Dynamic memory applications**.**