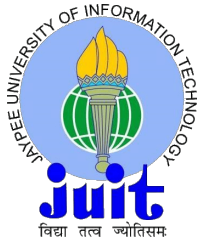


Data Structure

(Course Code: 25B11CI318)

Queue

Mr. Sandeep Kumar Patel

Assistant Professor

Jaypee University of Information Technology

Waknaghat, Solan, HP-173234

Introduction to Queue

Definition

A **Queue** is a linear data structure in which insertion is performed at one end called the **REAR**, and deletion is performed at the other end called the **FRONT**.

- Queue follows the **FIFO** principle.
- FIFO stands for **First In, First Out**.
- The element inserted first is removed first.
- Insertion takes place at the **REAR**.
- Deletion takes place at the **FRONT**.

Insertion $\rightarrow$ ENQUEUE

Deletion $\rightarrow$ DEQUEUE

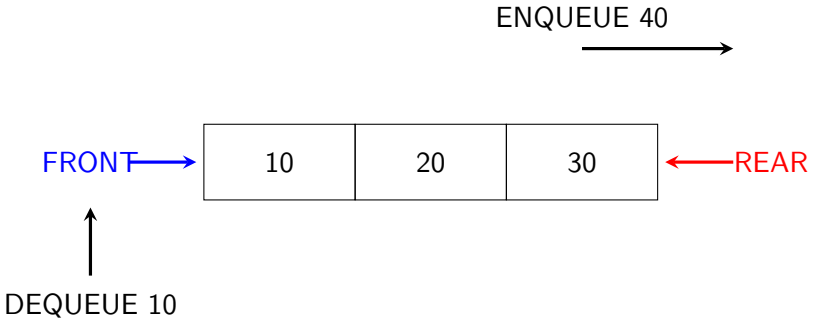
Queue: Real-Life Example

Example: Queue at a Ticket Counter

The person who arrives **first** is served **first**.

First In $\longrightarrow$ First Out

FIFO Principle



$10 \rightarrow 20 \rightarrow 30$

10 will be removed first

Abstract Data Type (ADT) of Queue

Queue ADT

A **Queue ADT** defines the data and operations of a queue without specifying how the queue is implemented.

Basic Queue Operations:

1. **Enqueue(x)** – Insert element x at the REAR.

Abstract Data Type (ADT) of Queue

Queue ADT

A **Queue ADT** defines the data and operations of a queue without specifying how the queue is implemented.

Basic Queue Operations:

1. **Enqueue(x)** – Insert element x at the REAR.
2. **Dequeue()** – Remove and return the FRONT element.

Abstract Data Type (ADT) of Queue

Queue ADT

A **Queue ADT** defines the data and operations of a queue without specifying how the queue is implemented.

Basic Queue Operations:

1. **Enqueue(x)** – Insert element x at the REAR.
2. **Dequeue()** – Remove and return the FRONT element.
3. **Front()** – Return the FRONT element without removing it.

Abstract Data Type (ADT) of Queue

Queue ADT

A **Queue ADT** defines the data and operations of a queue without specifying how the queue is implemented.

Basic Queue Operations:

1. **Enqueue(x)** – Insert element x at the REAR.
2. **Dequeue()** – Remove and return the FRONT element.
3. **Front()** – Return the FRONT element without removing it.
4. **isEmpty()** – Check whether the queue is empty.

Abstract Data Type (ADT) of Queue

Queue ADT

A **Queue ADT** defines the data and operations of a queue without specifying how the queue is implemented.

Basic Queue Operations:

1. **Enqueue(x)** – Insert element x at the REAR.
2. **Dequeue()** – Remove and return the FRONT element.
3. **Front()** – Return the FRONT element without removing it.
4. **isEmpty()** – Check whether the queue is empty.
5. **isFull()** – Check whether the queue is full (mainly for array implementation).

Queue ADT: Formal View

Queue ADT

A queue can be represented as:

$$Q = (D, O)$$

where:

- D = Set of data elements
- O = Set of operations

Operations:

Enqueue(Q, x)	$\rightarrow$ Insert x at REAR
Dequeue(Q)	$\rightarrow$ Delete FRONT element
Front(Q)	$\rightarrow$ Return FRONT element

Queue ADT vs Implementation

Queue ADT

Defines **WHAT** the queue does.

- Enqueue
- Dequeue
- Front
- isEmpty
- isFull

Implementation

Defines **HOW** the queue is stored.

- Using Array
- Using Linked List

Key Idea

ADT tells WHAT; Implementation tells HOW.

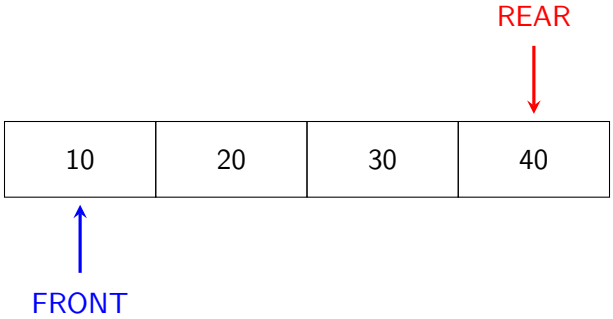
Basic Operations of Queue

Operation	Meaning	End Used
Enqueue(x)	Insert an element	REAR
Dequeue()	Delete an element	FRONT
Front()	View FRONT element	FRONT
isEmpty()	Check empty queue	–
isFull()	Check full queue	–

Important

Insertion takes place at the **REAR**, while deletion takes place at the **FRONT**.

Queue Representation



FRONT $\rightarrow$ 10 $\rightarrow$ 20 $\rightarrow$ 30 $\rightarrow$ 40 $\leftarrow$ REAR

Applications of Queue

Why Queue?

A queue is useful whenever elements need to be processed in the **First In, First Out (FIFO)** order.

Major applications of queues:

1. CPU scheduling
2. Printer scheduling
3. Operating system processes
4. Network packet management
5. Breadth First Search (BFS)
6. Keyboard and I/O buffering
7. Customer service systems
8. Simulation of real-world waiting lines

1: CPU Scheduling

Operating systems use queues to manage processes waiting for CPU execution.

Suppose processes arrive in the following order:

$$P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4$$

1: CPU Scheduling

Operating systems use queues to manage processes waiting for CPU execution.

Suppose processes arrive in the following order:

$$P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4$$

A simple scheduling system may process them as:

$$P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4$$

Example

In **First-Come, First-Served (FCFS)** scheduling, the process that arrives first gets CPU service first.

2: Printer Queue

When multiple users send documents to a printer, the documents can be placed in a queue.

$$D_1 \rightarrow D_2 \rightarrow D_3 \rightarrow D_4$$

2: Printer Queue

When multiple users send documents to a printer, the documents can be placed in a queue.

$$D_1 \rightarrow D_2 \rightarrow D_3 \rightarrow D_4$$

The printer processes:

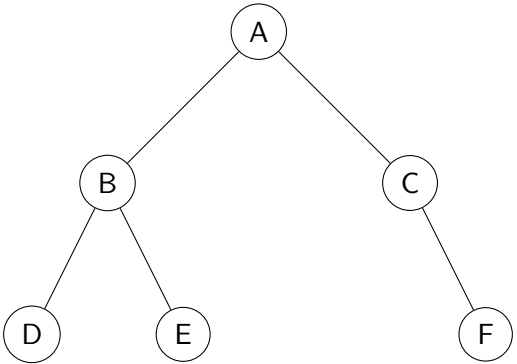
$$\boxed{D_1 \rightarrow D_2 \rightarrow D_3 \rightarrow D_4}$$

Key Idea

The document that enters the printing queue first is normally processed first.

3: Breadth First Search (BFS)

A queue is used in **Breadth First Search (BFS)** to explore a graph or tree level by level.



BFS uses a Queue

4: Network Packet Management

In computer networks, packets may arrive faster than they can be processed.

The packets can be temporarily stored in a queue:

$$P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4$$

4: Network Packet Management

In computer networks, packets may arrive faster than they can be processed.

The packets can be temporarily stored in a queue:

$$P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4$$

The network device processes packets from the FRONT:

$$\boxed{P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4}$$

Purpose

Queues help manage packets waiting for transmission or processing.

5: Keyboard and I/O Buffering

When data arrives faster than a device can process it, a queue can temporarily store the incoming data.

Example:

$$\text{Input} \rightarrow \boxed{\text{Queue}} \rightarrow \text{Processor}$$

5: Keyboard and I/O Buffering

When data arrives faster than a device can process it, a queue can temporarily store the incoming data.

Example:

$$\text{Input} \rightarrow \boxed{\text{Queue}} \rightarrow \text{Processor}$$

The processor processes the stored data in FIFO order.

Example

Keyboard input, printer data, disk requests, and communication buffers can use queues.

Applications of Queue: Summary

Application	Purpose of Queue
CPU Scheduling	Manage waiting processes
Printer Queue	Manage printing requests
BFS	Level-by-level graph traversal
Network Packets	Manage packets waiting for processing
I/O Buffering	Temporarily store incoming data
Customer Service	Manage waiting customers
Simulation	Model real-world waiting lines

Remember

Whenever a problem naturally requires **First In, First Out**, think of a **Queue**.

Queue Using Two Stacks

Idea

A Queue follows **FIFO**, while a Stack follows **LIFO**. We can implement a Queue using two Stacks.

Let the two stacks be:

S_1 S_2

- **Enqueue:** Push the element into S_1.
- **Dequeue:** Pop from S_2.
- If S_2 is empty, transfer all elements from S_1 to S_2.

Key Idea

Two LIFO operations can be combined to produce FIFO behavior.

Queue Using Two Stacks: Example

Perform:

$$Enqueue(10),\ Enqueue(20),\ Enqueue(30)$$

After insertion:

$$S_1 : \boxed{30} \rightarrow \boxed{20} \rightarrow \boxed{10}$$

$$S_2 : \boxed{\text{Empty}}$$

Queue Using Two Stacks: Example

Perform:

$$Enqueue(10),\ Enqueue(20),\ Enqueue(30)$$

After insertion:

$$S_1 : \boxed{30} \rightarrow \boxed{20} \rightarrow \boxed{10}$$

$$S_2 : \boxed{\text{Empty}}$$

For `Dequeue()`, transfer elements:

$$S_1 \longrightarrow S_2$$

$$S_2 : \boxed{10} \rightarrow \boxed{20} \rightarrow \boxed{30}$$

Queue Using Two Stacks: Example

Perform:

$$Enqueue(10),\ Enqueue(20),\ Enqueue(30)$$

After insertion:

$$S_1 : \boxed{30} \rightarrow \boxed{20} \rightarrow \boxed{10}$$

$$S_2 : \boxed{\text{Empty}}$$

For `Dequeue()`, transfer elements:

$$S_1 \longrightarrow S_2$$

$$S_2 : \boxed{10} \rightarrow \boxed{20} \rightarrow \boxed{30}$$

Queue Using Two Stacks: Enqueue

```
1 void enqueue(int value)
2 {
3     push(&s1, value);
4 }
```

Queue Using Two Stacks: Enqueue

```
1 void enqueue(int value)
2 {
3     push(&s1, value);
4 }
```

Time Complexity $= O(1)$

Queue Using Two Stacks: Dequeue

```
int dequeue()
{
    int value;

    if(isEmpty(s1) && isEmpty(s2))
    {
        printf("Queue Underflow");
        return -1;
    }

    if(isEmpty(s2))
    {
        while(!isEmpty(s1))
        {
            push(&s2, pop(&s1));
        }
    }
```

Queue Using Two Stacks: Dequeue

```
int dequeue()
{
    int value;

    if(isEmpty(s1) && isEmpty(s2))
    {
        printf("Queue Underflow");
        return -1;
    }

    if(isEmpty(s2))
    {
        while(!isEmpty(s1))
        {
            push(&s2, pop(&s1));
        }
    }
```

Complete Queue Using Two Stacks

```
#define MAX 100

int s1[MAX], s2[MAX];
int top1 = -1, top2 = -1;

void push(int stack[], int *top, int value)
{
    stack[++(*top)] = value;
}

int pop(int stack[], int *top)
{
    return stack[(*top)--];
}

void enqueue(int value)
{
```

Stack Using Two Queues

Idea

A Stack follows **LIFO**, while a Queue follows **FIFO**. We can implement a Stack using two Queues.

Let the two queues be:

$\boxed{Q_1}$ $\boxed{Q_2}$

- Insert the new element into Q_2.
- Move all elements from Q_1 to Q_2.
- Swap Q_1 and Q_2.
- Now the newest element is at the FRONT of Q_1.

Key Idea

The Queue is rearranged so that the newest element always appears at the FRONT.

Stack Using Two Queues: Push

Initially:

$$Q_1 = [10,\ 20]$$

Push **30**.

Stack Using Two Queues: Push

Initially:

$$Q_1 = [10,\ 20]$$

Push **30**.

First insert 30 into Q_2:

$$Q_2 = [30]$$

Stack Using Two Queues: Push

Initially:

$$Q_1 = [10,\ 20]$$

Push **30**.
First insert 30 into Q_2:

$$Q_2 = [30]$$

Move all elements from Q_1 to Q_2:

$$Q_2 = [30,\ 10,\ 20]$$

Stack Using Two Queues: Push

Initially:

$$Q_1 = [10,\ 20]$$

Push **30**.
First insert 30 into Q_2:

$$Q_2 = [30]$$

Move all elements from Q_1 to Q_2:

$$Q_2 = [30,\ 10,\ 20]$$

Swap Q_1 and Q_2:

$$Q_1 = [30,\ 10,\ 20]$$

Therefore:

Stack Using Two Queues: Push

```
void push(int value)
{
    enqueue(&q2, value);

    while(!isEmpty(q1))
    {
        enqueue(&q2, dequeue(&q1));
    }

    swap(&q1, &q2);
}
```

Stack Using Two Queues: Push

```
void push(int value)
{
    enqueue(&q2, value);

    while(!isEmpty(q1))
    {
        enqueue(&q2, dequeue(&q1));
    }

    swap(&q1, &q2);
}
```

Time Complexity $= O(n)$

Stack Using Two Queues: Pop

```
int pop()
{
    if(isEmpty(q1))
    {
        printf("Stack Underflow");
        return -1;
    }

    return dequeue(&q1);
}
```

Stack Using Two Queues: Pop

```
int pop()
{
    if(isEmpty(q1))
    {
        printf("Stack Underflow");
        return -1;
    }

    return dequeue(&q1);
}
```

Time Complexity $= O(1)$

Complete Stack Using Two Queues

```
#define MAX 100

int q1[MAX], q2[MAX];
int front1 = 0, rear1 = -1;
int front2 = 0, rear2 = -1;

void push(int value)
{
    q2[++rear2] = value;

    while(front1 <= rear1)
    {
        q2[++rear2] = q1[front1++];
    }

    int *temp;
```

Thank You

Questions?