

Queues

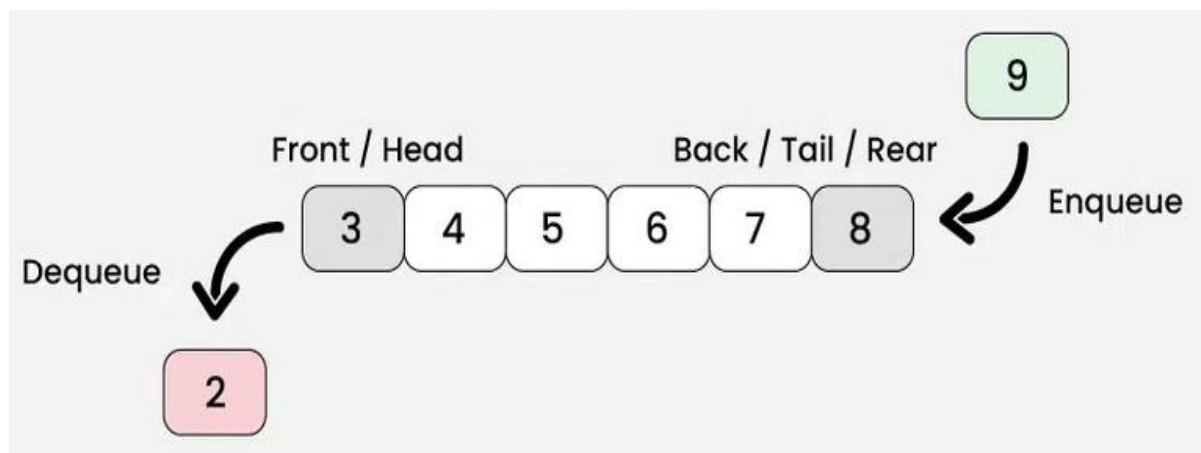
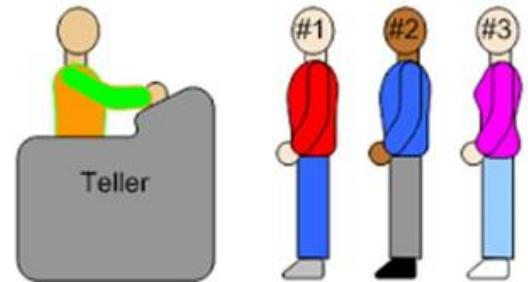
PART-3

Queues: Types of Queues- Circular Queue, Deque and operations.

Queue is a Linear Data Structure that follows the **FIFO (First-In, First-Out)** principle.

This means the first element added to the queue is the first one to be removed.

Example: It's like a line of people waiting to buy tickets: the person who arrives first is the first to be served.



Queue Operations

Enqueue(x)	Insert element at the rear.
Deque()	Remove element from the front.
Peek/Front()	Get front element without removing.
isEmpty()	Check if queue is empty.
isFull()	Check if queue is full (in array implementation).

1. Enqueue (Insert an element into Queue)

Algorithm:

1. Check if the queue is full.
 - If $\text{rear} == \text{SIZE} - 1$ (for linear queue), then **Overflow**.
 2. If $\text{front} == -1$, set $\text{front} = 0$ (first insertion).
 3. Increment rear by 1.
 4. Insert the new element at $\text{queue}[\text{rear}]$.
-

2. Dequeue (Delete an element from Queue)

Algorithm:

1. Check if the queue is empty.
 - If $\text{front} == -1$ or $\text{front} > \text{rear}$, then **Underflow**.
 2. Delete the element at $\text{queue}[\text{front}]$.
 3. Increment front by 1.
 4. If after deletion, $\text{front} > \text{rear}$, reset $\text{front} = \text{rear} = -1$ (queue becomes empty).
-

3. Peek (View the front element without removing it)

Algorithm:

1. Check if the queue is empty.
 - If $\text{front} == -1$ or $\text{front} > \text{rear}$, print **Queue is empty**.
 2. Else, return $\text{queue}[\text{front}]$.
-

4. Display (Show all elements in Queue)

Algorithm:

1. Check if the queue is empty.
 - If $\text{front} == -1$ or $\text{front} > \text{rear}$, print **Queue is empty**.
2. Else, traverse from front to rear, and print each element.

Representation of Queues

Queues can be represented using various underlying data structures. The most common implementations are using arrays or linked lists.

1. Array-Based Representation

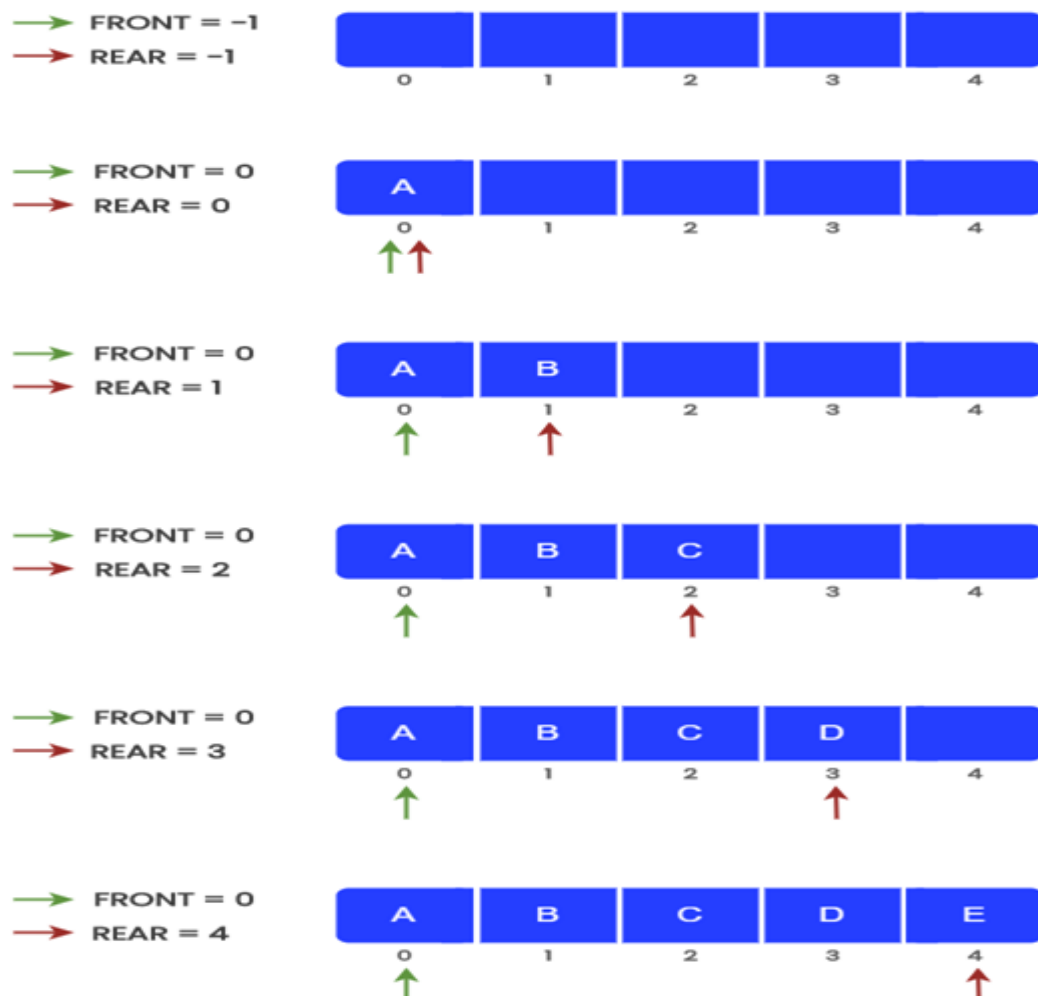
An array can be used to implement a queue. Two pointers or indices, typically called front and rear, are used to keep track of the first and last elements.

- **Enqueue:** The new element is added at the rear, and the rear index is incremented.

Algorithm:

1. Check if the queue is full.
 - If $\text{rear} == \text{SIZE} - 1$ (for linear queue), then **Overflow**.
2. If $\text{front} == -1$, set $\text{front} = 0$ (first insertion).
3. Increment rear by 1.
4. Insert the new element at $\text{queue}[\text{rear}]$.

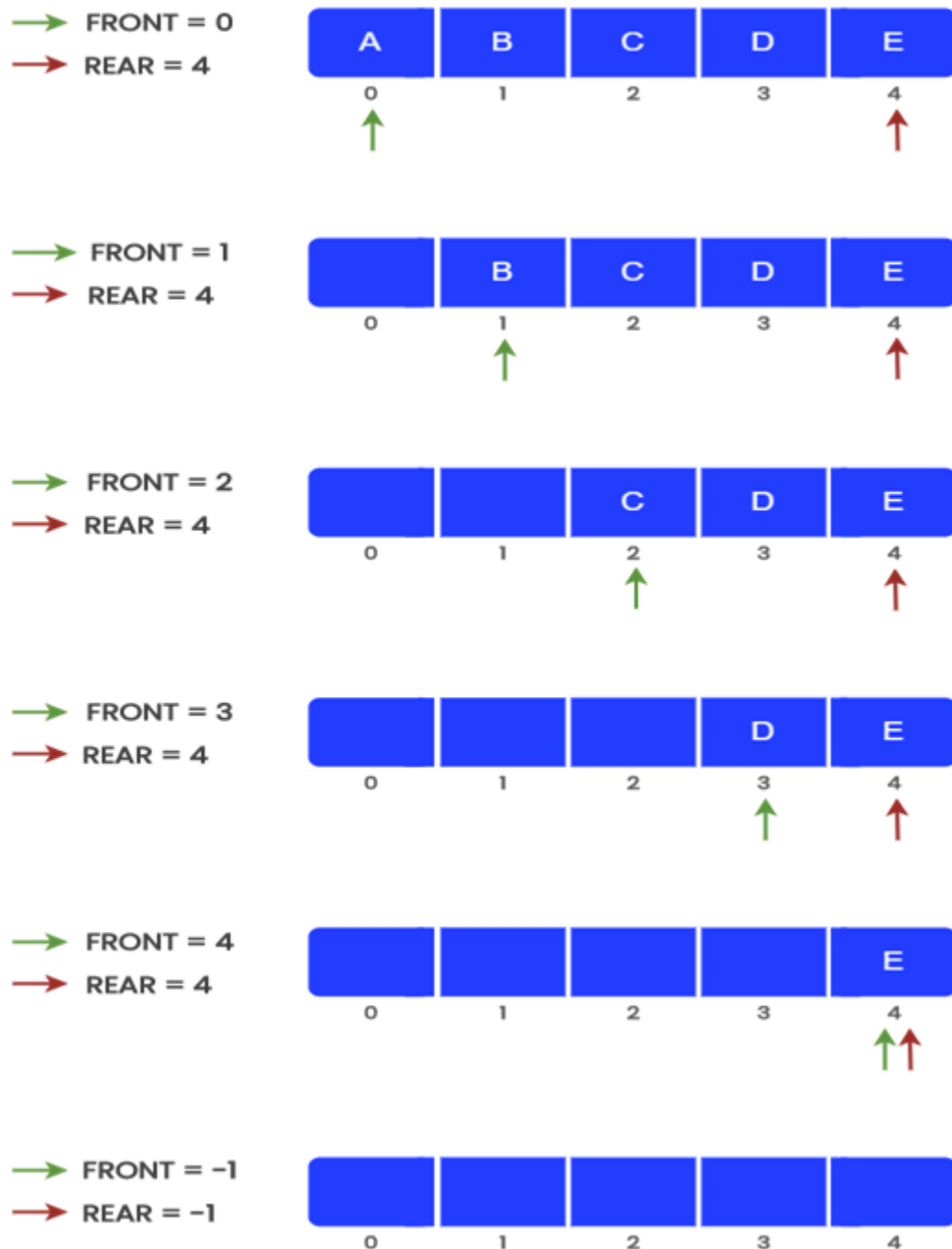
Consider a queue of size 5. Suppose we have to insert A, B, C, D and E into the queue. The following changes will be made to the queue.



- **Dequeue:** The element at the front is removed, and the front index is incremented.

Algorithm:

1. Check if the queue is empty.
 - If $\text{front} == -1$ or $\text{front} > \text{rear}$, then **Underflow**.
2. Delete the element at $\text{queue}[\text{front}]$.
3. Increment front by 1.
4. If after deletion, $\text{front} > \text{rear}$, reset $\text{front} = \text{rear} = -1$ (queue becomes empty).



A potential issue with this approach is that as elements are dequeued, the front index moves forward, leaving empty space at the beginning of the array. This can lead to a **queue overflow** even if the array is not full.

A **circular queue**, which wraps the indices around to the beginning of the array when they reach the end, is a common solution to this problem.

2. Linked List-Based Representation

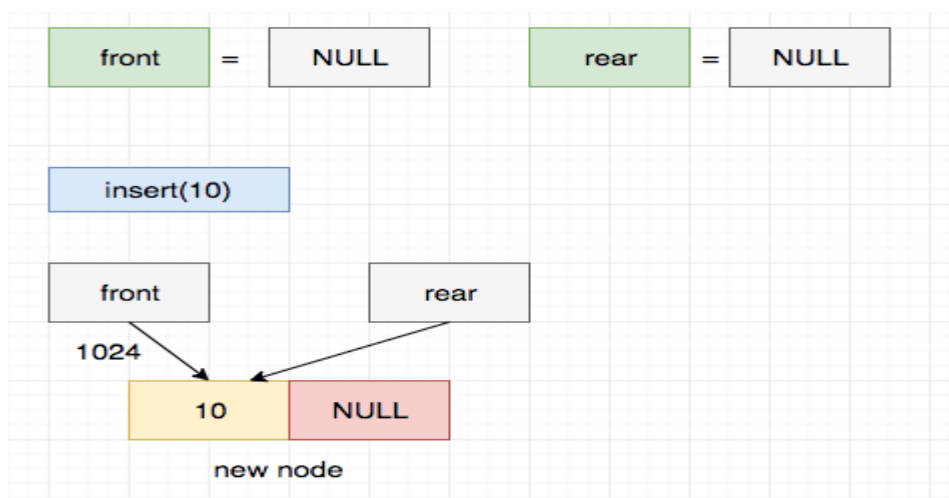
A linked list is a more dynamic and flexible way to implement a queue. The head of the list serves as the front of the queue, and the tail serves as the rear.

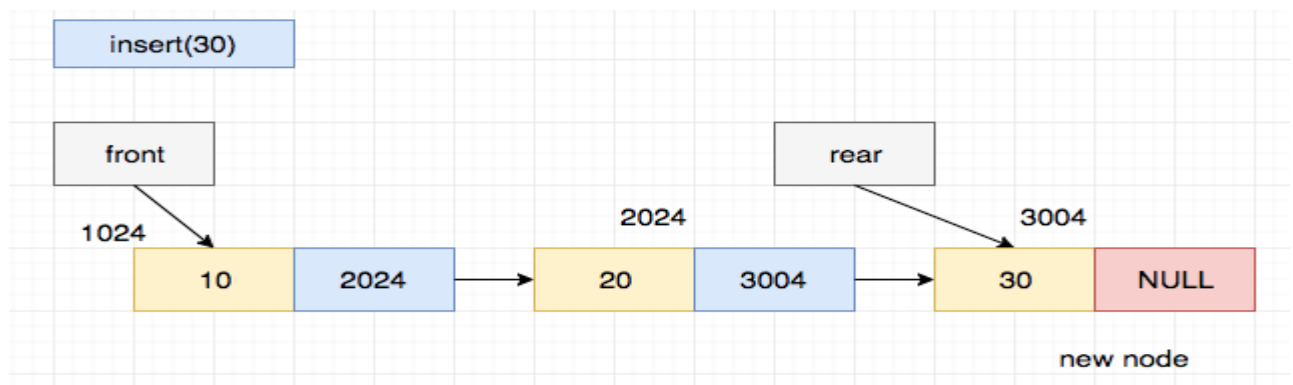
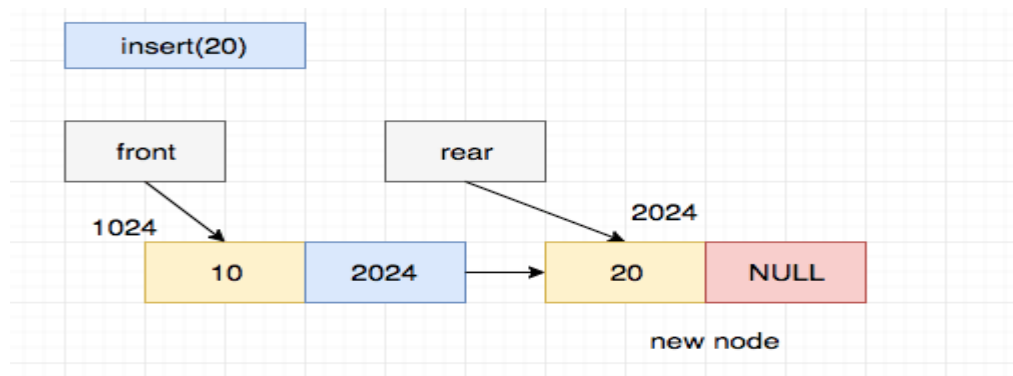
- **Enqueue:** A new node is added to the tail of the linked list. This is an efficient operation.

ENQUEUE (Insert at Rear)

Algorithm:

1. Create a new node with given value.
2. If memory not available → **overflow**.
3. If queue is empty (front == NULL && rear == NULL):
 - Set both front and rear to the new node.
4. else:
 - Link rear->next = newNode;
 - Move rear = newNode;



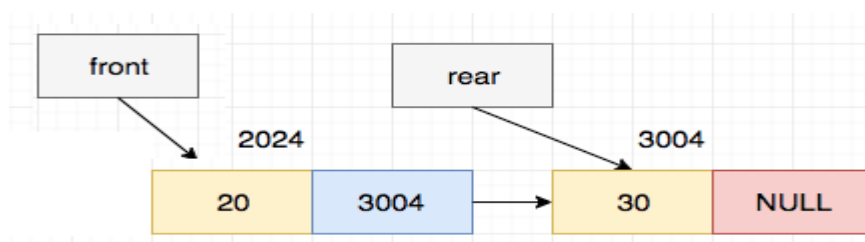


- **Dequeue:** The head node is removed. The next node in the list becomes the new head. This is also an efficient operation.

DEQUEUE (Delete from Front)

Algorithm:

1. If $\text{front} == \text{NULL}$ \rightarrow Queue is empty (underflow).
2. else:
 - Store $\text{temp} = \text{front}$.
 - Move $\text{front} = \text{front} \rightarrow \text{next}$.
 - If $\text{front} == \text{NULL}$ after update \rightarrow set $\text{rear} = \text{NULL}$ (queue became empty).
 - Free the old front ($\text{free}(\text{temp})$).



This implementation avoids the fixed-size limitations of arrays and can grow or shrink dynamically as needed, limited only by available memory.

Types of Queues

There are four main types of queues, each with specific characteristics that define how elements are added and removed.

1. Simple Queue
2. Circular Queue
3. Priority Queue
4. Double-Ended Queue (Deque)

1. Simple Queue

This is the most basic type, following the **FIFO (First-In, First-Out)** principle. Elements are enqueued at the rear and dequeued from the front. It operates like a single-lane queue at a checkout counter.

Limitation of Linear Queue

Look at the following queue.



You cannot insert a new value now, because the queue is completely full. Consider the following case: two consecutive deletions are made.

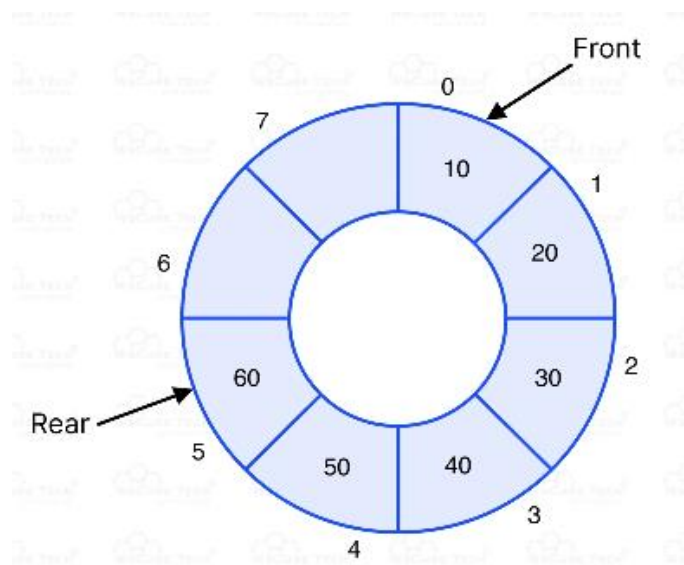


Even though there is enough space to hold a new element, the overflow condition still occurs since the condition $\text{rear} = \text{MAX} - 1$ is still valid. This non-usable empty space is a major drawback of a linear queue. A solution to this problem is to make the queue circular.

2. Circular Queue

A circular queue is an improved version of a simple queue. It uses a fixed-size array but avoids the issue of wasted space that occurs when elements are dequeued. When the rear of the array is reached, new elements can be enqueued at the beginning of the array, effectively "wrapping around."

A circular queue is a special type of queue where the last element is connected back to the first element thus forming a circle. In the circular queue, the first index comes right after the last index.



We start insertion from the beginning of the queue if we reach the end of the queue.

1. Enqueue (Insert element)

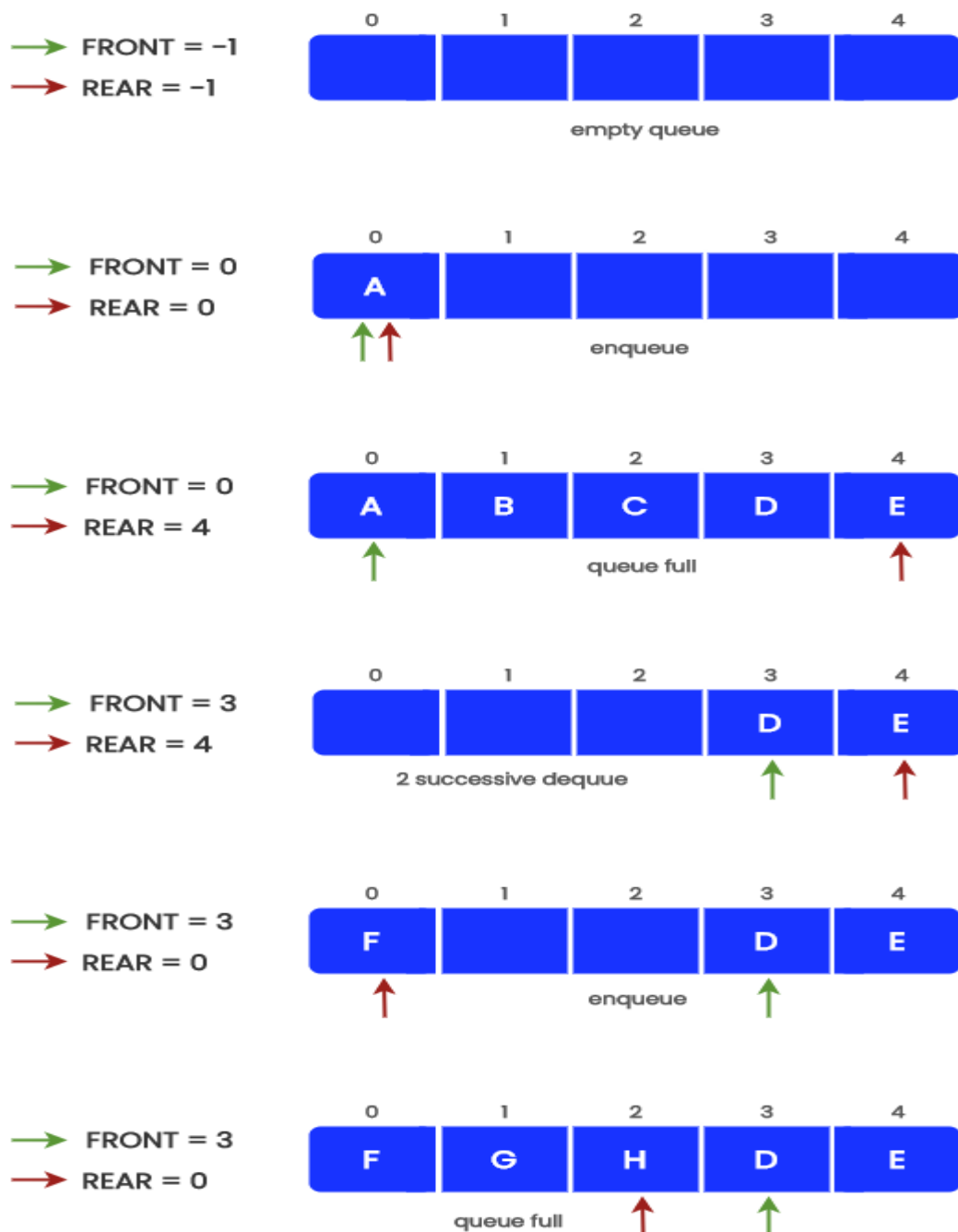
Steps:

1. Check if queue is **full**. If yes → Overflow.
2. If empty → $\text{front} = \text{rear} = 0$.
3. Otherwise → $\text{move rear} = (\text{rear} + 1) \% \text{SIZE}$.
4. Insert element at $\text{queue}[\text{rear}]$.

2. Dequeue (Remove element)

Steps:

1. Check if queue is **empty**. If yes → Underflow.
2. Store element at $\text{queue}[\text{front}]$.
3. If only one element → reset $\text{front} = \text{rear} = -1$.
4. Otherwise → $\text{move front} = (\text{front} + 1) \% \text{SIZE}$.



Applications of Circular Queues

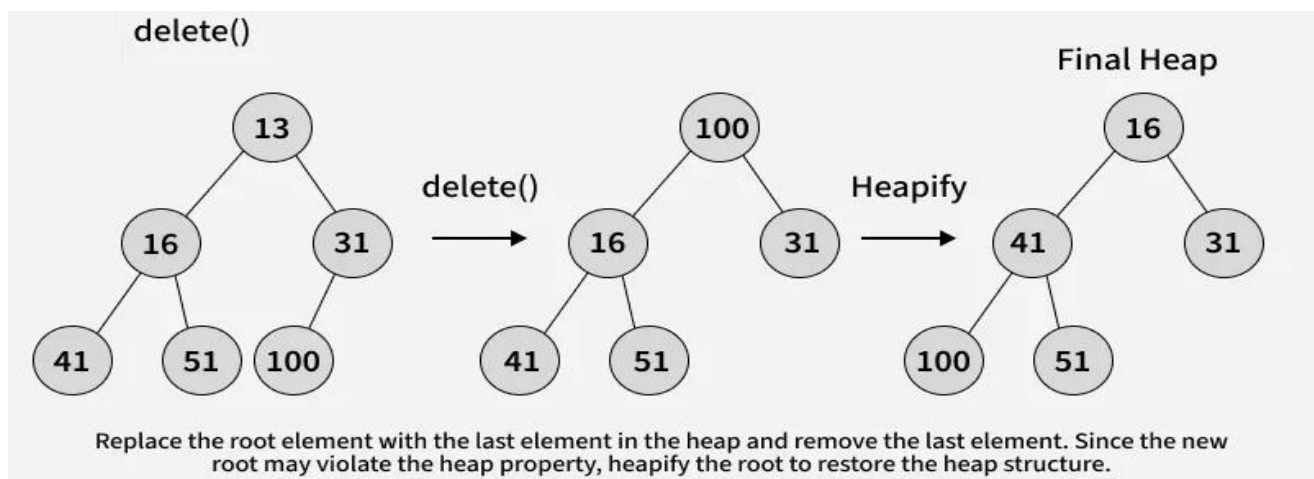
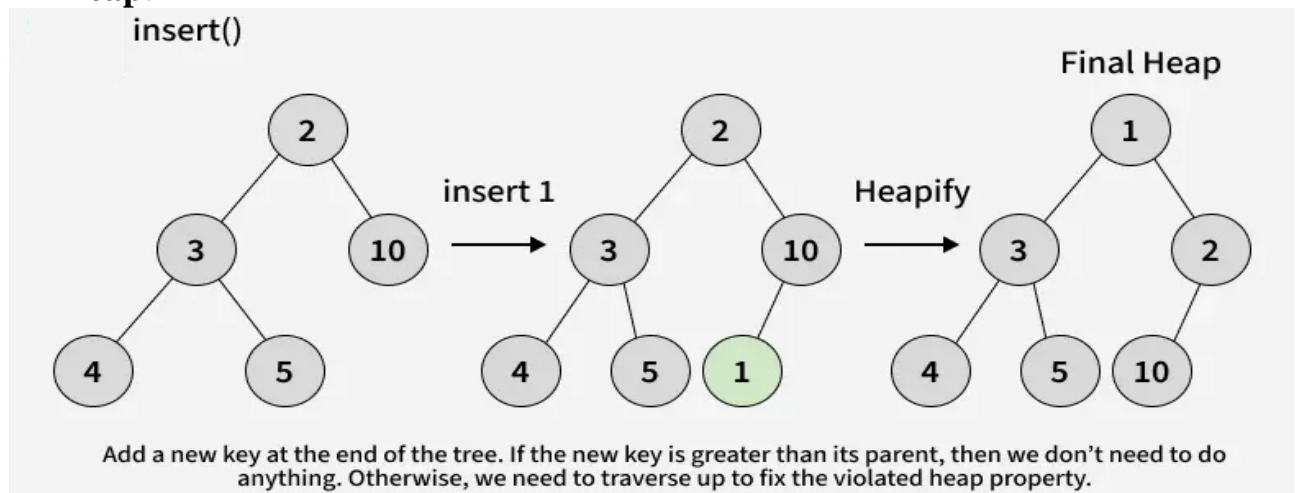
1. CPU Scheduling (Round-Robin Algorithm)
2. Buffer Management (Circular Buffers)
3. Traffic Signal Management
4. Printer Queue (Spooling)
6. Call Center Systems
7. Multiplayer Games / Round-based Games
8. Music / Playlist Management

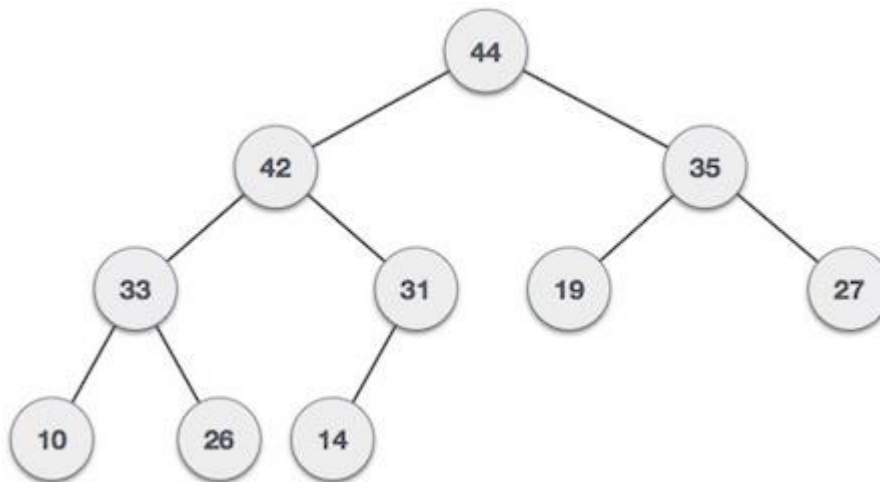
3. Priority Queue

A priority queue is different because the **order of removal is based on an element's priority**, not its arrival time.

- Elements with higher priority are dequeued before elements with lower priority.
- If two elements have the same priority, their order of removal is determined by their arrival order.
- Priority queues are most efficiently implemented using a **heap** data structure. A heap is a specialized tree-based data structure that satisfies the heap property:
 - **Min-Heap:** The value of each node is less than or equal to the values of its children. This is used for a priority queue where a **lower** value indicates a **higher** priority.
 - **Max-Heap:** The value of each node is greater than or equal to the values of its children. This is used for a priority queue where a **higher** value indicates a **higher** priority.

Min Heap:



Max Heap:**Applications**

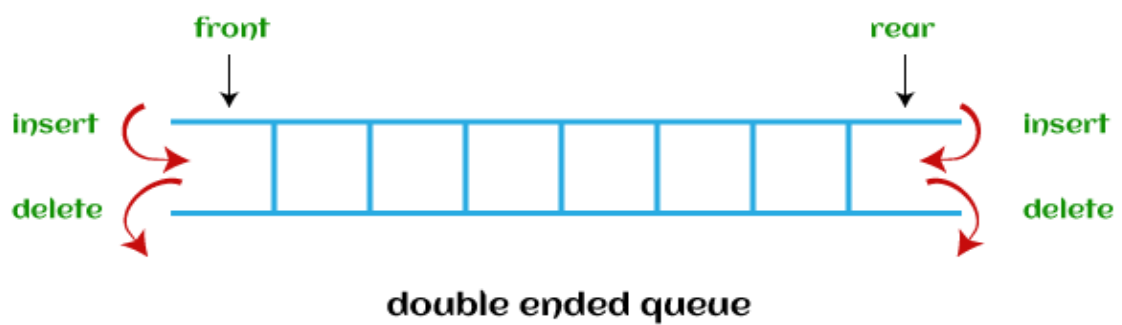
Priority queues are used in many real-world scenarios:

- **Dijkstra's Algorithm:** Used to find the shortest path in a graph. The priority queue stores the unvisited vertices, prioritizing the one with the smallest distance from the source.
- **Operating Systems:** Used for CPU scheduling. Processes are placed in a priority queue, and the one with the highest priority is given access to the CPU.
- **Event Simulation:** In discrete event simulations, a priority queue is used to manage and process events in chronological order.
- **Data Compression:** Algorithms like Huffman coding use priority queues to build optimal prefix codes.

4. Double-Ended Queue (Deque)

A deque, pronounced "deck," is a more flexible queue that allows elements to be added or removed from **both the front and the rear**. It doesn't follow the strict FIFO or LIFO rule.

- It's like a hybrid of a queue and a stack.
- Operations like `addFirst()`, `addLast()`, `removeFirst()`, and `removeLast()` are supported.



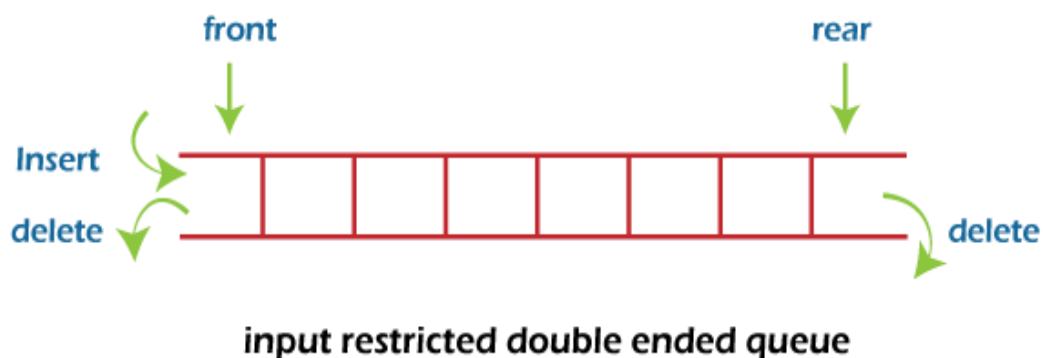
Types of Double Ended Queue

There are two types of deque:

- Input restricted queue
- Output restricted queue

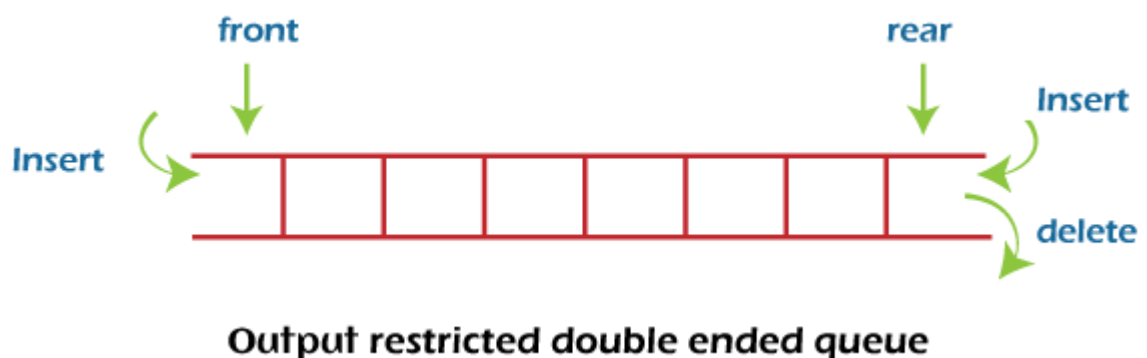
Input restricted Queue

In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



Output restricted Queue

In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Operations on Deque

1. **InsertFront(x)** → Insert element at front.
2. **InsertRear(x)** → Insert element at rear.
3. **DeleteFront()** → Remove element from front.
4. **DeleteRear()** → Remove element from rear.
5. **GetFront()** → Peek front element.
6. **GetRear()** → Peek rear element.
7. **isEmpty()** → Check if deque is empty.
8. **isFull()** → Check if deque is full.

Applications of Deque

1. **Browser history** (back & forward navigation).
2. **Undo/Redo operations** in editors.
3. **Job scheduling** (add/remove tasks from both ends).
4. **Palindrome checking** (characters compared from both ends).

Applications of Queues

1. Operating Systems

Queues are vital for managing tasks in operating systems.

- **CPU Scheduling:** Processes waiting for CPU time are held in a queue. The operating system's scheduler picks the next process from the front of the queue to give it CPU access.
- **Disk Scheduling:** Requests to read from or write to a disk are often managed in a queue to optimize disk head movement.
- **Interrupt Handling:** When multiple devices (like a keyboard, mouse, or printer) request a service, their requests are placed in a queue and processed in the order they were received.

2. Networking and Communications

Queues play a critical role in data transmission and message handling.

- **Packet Transmission:** Data packets traveling over a network are often stored in a queue (a buffer) at a router or switch. This ensures that packets are processed in the order they arrive, preventing network congestion and data loss.
- **Email and Message Queues:** Email servers use queues to hold outgoing messages. If a server is busy, messages are queued and sent when the server becomes available. Message queuing systems are also widely used in distributed applications for asynchronous communication.

3. Data Buffering

Queues are used to handle temporary data storage when there's a difference in speed between data producers and consumers.

- **Streaming Media:** A queue is used as a buffer when streaming video or audio. The data is buffered to ensure smooth playback, even if the network connection is inconsistent.
- **I/O Operations:** Data from input devices (like a keyboard or file) and data going to output devices (like a printer) are often held in a queue to manage the flow of information efficiently.

4. Other Applications

- **Breadth-First Search (BFS):** This graph traversal algorithm uses a queue to explore all the nodes at the current depth level before moving to the next level.
- **Print Spooling:** When multiple documents are sent to a printer, they are placed in a print queue. The printer then processes them one by one in the order they were received.
- **Call Center Systems:** Customers waiting for a representative are placed in a queue. The first customer to call is the first one to be connected to a free agent.
- **Simulations:** Queues are used in simulations to model real-world waiting lines and analyze system performance, such as traffic flow at an intersection or customers at a bank.

1. C Programs to implement Queue Operations using Arrays

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 5

int queue[SIZE];
int front = -1, rear = -1;

// Enqueue operation
void enqueue(int value)
{
    if (rear == SIZE - 1)
    {
        printf("Queue Overflow! Cannot enqueue %d\n", value);
        return;
    }
    if (front == -1) front = 0;
    queue[++rear] = value;
    printf("Enqueued %d to Queue.\n", value);
}

// Dequeue operation
void dequeue()
{
    if (front == -1 || front > rear)
    {
        printf("Queue Underflow! Cannot dequeue.\n");
        return;
    }
    printf("Dequeued %d from Queue.\n", queue[front++]);
    // Reset front and rear if queue becomes empty
    if (front > rear)
    {
        front = rear = -1;
    }
}
```

// Peek operation

```
void peek()
{
    if (front == -1 || front > rear)
    {
        printf("Queue is Empty.\n");
        return;
    }
    printf("Front element is %d\n", queue[front]);
}
```

// Display operation

```
void display()
{
    int i;
    if (front == -1 || front > rear)
    {
        printf("Queue is Empty.\n");
        return;
    }
    printf("Queue elements (FRONT to REAR):\n");
    for (i = front; i <= rear; i++)
    {
        printf("%d ", queue[i]);
    }
    printf("\n");
}
```

// Main function with menu

```
int main()
{
    int choice, value;
    while (1)
    {
        printf("\n--- Queue using Array ---\n");
        printf("1. Enqueue\n2. Dequeue\n3. Peek\n4. Display\n5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
```



```
switch (choice)
{
case 1:
    printf("Enter value to Enqueue: ");
    scanf("%d", &value);
    enqueue(value);
    break;

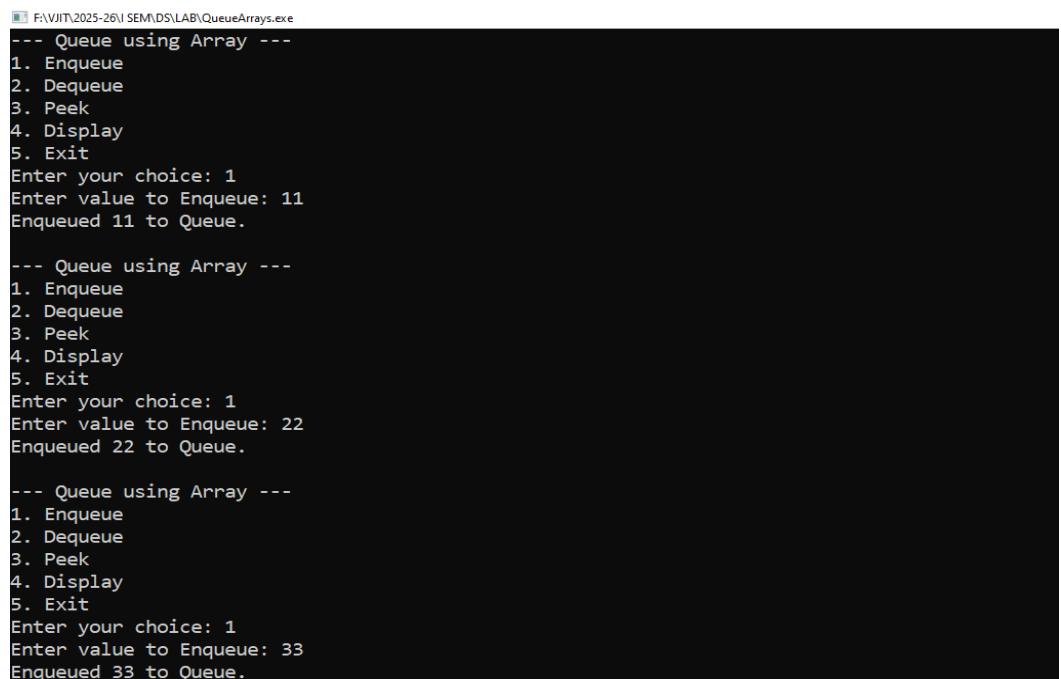
case 2:
    dequeue();
    break;

case 3:
    peek();
    break;

case 4:
    display();
    break;

case 5:
    printf("Exiting.\n");
    exit(0);

default:
    printf("Invalid choice! Try again.\n");
}
}
return 0;
}
```

Output:

```
F:\VJIT\2025-26\I SEM\DS\LAB\QueueArrays.exe
--- Queue using Array ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to Enqueue: 11
Enqueued 11 to Queue.

--- Queue using Array ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to Enqueue: 22
Enqueued 22 to Queue.

--- Queue using Array ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to Enqueue: 33
Enqueued 33 to Queue.
```

F:\VJIT\2025-26\I SEM\DS\LAB\QueueArrays.exe

```
--- Queue using Array ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 4
Queue elements (FRONT to REAR):
11 22 33
```

```
--- Queue using Array ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 3
Front element is 11
```

```
--- Queue using Array ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 2
Dequeued 11 from Queue.
```

F:\VJIT\2025-26\I SEM\DS\LAB\QueueArrays.exe

```
--- Queue using Array ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 4
Queue elements (FRONT to REAR):
22 33
```

```
--- Queue using Array ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 5
Exiting.
```

```
-----
Process exited after 79.88 seconds with return value 0
Press any key to continue . . . █
```

2. C Programs to implement Queue Operations using Linked List

```
#include <stdio.h>
#include <stdlib.h>

// Define structure for a queue node
struct Node
{
    int data;
    struct Node* next;
};

// Initialize front and rear pointers
struct Node* front = NULL;
struct Node* rear = NULL;

// Enqueue operation
void enqueue(int value)
{
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode)
    {
        printf("Heap overflow\n");
        return;
    }
    newNode->data = value;
    newNode->next = NULL;
    if (rear == NULL)
    {
        // Queue is empty
        front = rear = newNode;
    } else {
        rear->next = newNode;
        rear = newNode;
    }
    printf("Enqueued %d into Queue\n", value);
}
```

// Dequeue operation

```
void dequeue()
{
    if (front == NULL)
    {
        printf("Queue underflow\n");
        return;
    }
    struct Node* temp = front;
    printf("Dequeued %d from Queue\n", front->data);
    front = front->next;
    if (front == NULL)
    {
        rear = NULL;
    }
    free(temp);
}
```

// Peek operation

```
void peek()
{
    if (front == NULL)
    {
        printf("Queue is empty\n");
        return;
    }
    printf("Front element is %d\n", front->data);
}
```

// Display queue

```
void display()
{
    struct Node* temp = front;
    if (temp == NULL)
    {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements: ");
    while (temp != NULL)
    {
        printf("%d <- ", temp->data);
        temp = temp->next;
    }
}
```

```
printf("NULL\n");
}
// Main function to test queue operations
int main()
{
    int choice, value;
    while (1)
    {
        printf("\n--- Queue Menu ---\n");
        printf("1. Enqueue\n2. Dequeue\n3. Peek\n4. Display\n5. Exit\n");
        printf("Enter your Choice: ");
        scanf("%d", &choice);
        switch (choice)
        {
            case 1:
                printf("Enter value to Enqueue: ");
                scanf("%d", &value);
                enqueue(value);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                peek();
                break;
            case 4:
                display();
                break;
            case 5:
                printf("Exit from the Program\n");
                exit(0);
            default:
                printf("Invalid Choice\n");
        }
    }
    return 0;
}
```

Output:

F:\VJIT\2025-26\I SEM\DS\LAB\QueueLL.exe

```
--- Queue Menu ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your Choice: 1
Enter value to Enqueue: 44
Enqueued 44 into Queue

--- Queue Menu ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your Choice: 1
Enter value to Enqueue: 55
Enqueued 55 into Queue

--- Queue Menu ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your Choice: 4
Queue elements: 44 <- 55 <- NULL
```

F:\VJIT\2025-26\I SEM\DS\LAB\QueueLL.exe

```
--- Queue Menu ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your Choice: 3
Front element is 44

--- Queue Menu ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your Choice: 2
Dequeued 44 from Queue

--- Queue Menu ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your Choice: 4
Queue elements: 55 <- NULL
```

F:\VJIT\2025-26\I SEM\DS\LAB\QueueLL.exe

```
--- Queue Menu ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your Choice: 5
Exit from the Program

-----
Process exited after 8.692 seconds with return value 0
Press any key to continue . . .
```