SYLLABUS

1.) Queues: Introduction to queues:

2.) properties and operations,

3.) implementing queues using arrays and linked lists,

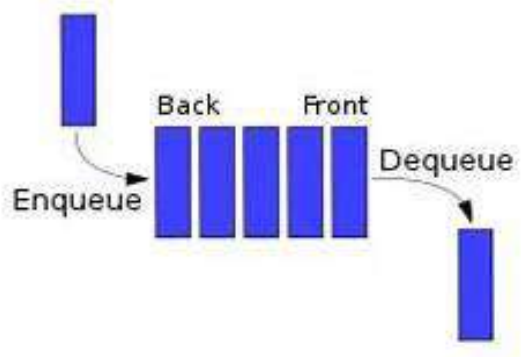
Applications of queues in breadth-first search,

4.) scheduling, etc. Deques:

5.) Introduction to deques (double-ended queues),

6.) Operations on deques and their applications.

1.) Introduction to Queues



- A queue is a linear data structure that follows the **First-In-First-Out (FIFO)** principle.
- In a queue, elements are stored in a specific order, and the operations are performed as follows:
    - **Enqueue (Insertion)**: Adds an element to the rear of the queue.
    - **Dequeue (Removal)**: Removes the front element from the queue.
- Think of a queue like a line of people waiting to purchase tickets, where the first person in line is the first person served (first come, first serve).

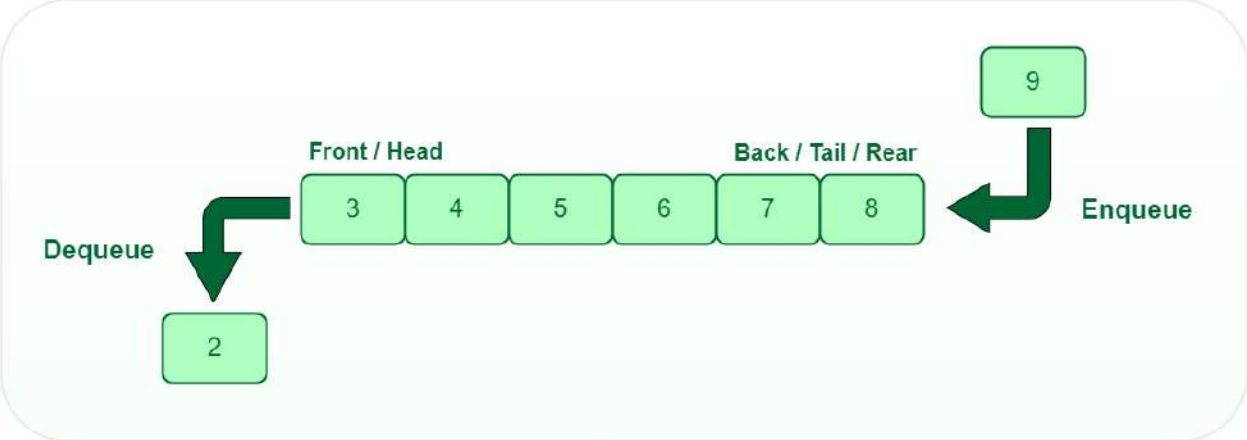
Queue Characteristics

1. **FIFO Property**:
    - The element that is first pushed into the queue is the first one to be removed.
    - The front of the queue is the position ready to be served, and the rear is the most recently added position.
2. **Queue Representation**:
    - Queues can be represented using either arrays or linked lists.
    - In an array-based representation, we use the front and rear indices to manage the queue.

- In a linked list representation, we maintain pointers to the front and rear nodes.

Queues are used in various applications, including process scheduling, printer spooling, and managing data efficiently! □□

2.) Queue Properties



1. **FIFO (First-In-First-Out)**:
   - The element that is first enqueued (inserted) will be the first one dequeued (removed).
   - Think of a queue like a line of people waiting in a queue, where the person who arrives first is served first.
2. **Front and Rear**:
   - The **front** of the queue is the position where elements are dequeued.
   - The **rear** of the queue is the position where elements are enqueued.

Basic Queue Operations

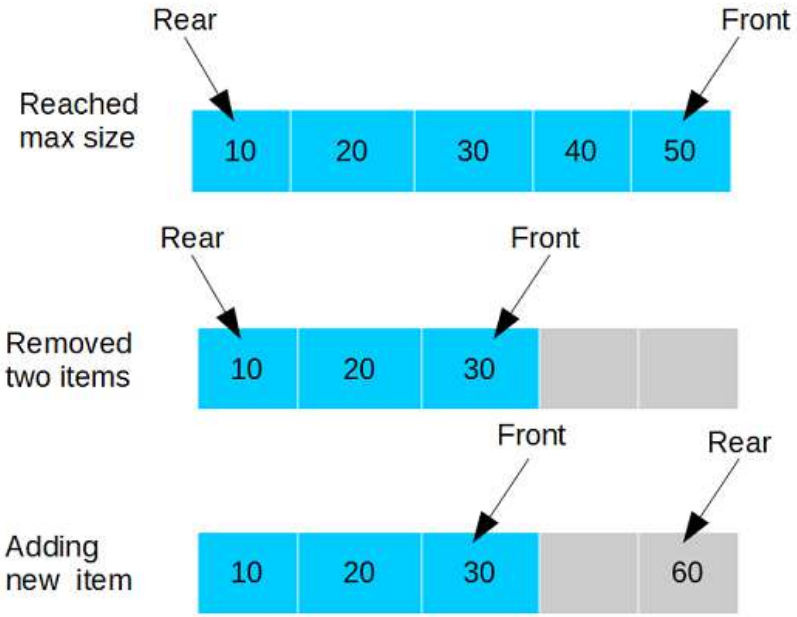
1. **Enqueue (Insertion)**:
   - Adds an element to the rear of the queue.
   - If the queue is full, it results in an **Overflow** condition.

2. **Dequeue (Removal)**:
    - Removes the front element from the queue.
    - If the queue is empty, it results in an **Underflow** condition.
3. **Front Element**:
    - Returns the front element of the queue without removing it.
4. **isEmpty**:
    - Returns true if the queue is empty, else false.

Remember, queues are used in various applications, including process scheduling, printer spooling, and managing data efficiently! □□

3.) Queue Using Arrays

- In an array-based implementation of a queue, we use an array to store the elements.
- We maintain two pointers: one for the front (where elements are dequeued) and another for the rear (where elements are enqueued).
- Here's an example of implementing a queue using an array in C++:



```
#include <iostream>
using namespace std;

class Queue {
private:
    int* arr;
    int front;
    int rear;
    int capacity;

public:
    Queue(int size) {
        capacity = size;
        arr = new int[capacity];
        front = rear = -1;
    }

    bool isEmpty() {
        return front == -1;
    }

    bool isFull() {
        return (rear + 1) % capacity == front;
    }

    void enQueue(int item) {
        if (isFull()) {
            cout << "Queue is full. Cannot enqueue.\n";
            return;
        }
        if (isEmpty())
            front = 0;
        rear = (rear + 1) % capacity;
        arr[rear] = item;
    }
```

```
    void deQueue() {
        if (isEmpty()) {
            cout << "Queue is empty. Cannot dequeue.\n";
            return;
        }
        if (front == rear)
            front = rear = -1;
        else
            front = (front + 1) % capacity;
    }

    int getFront() {
        if (isEmpty()) {
            cout << "Queue is empty.\n";
            return -1;
        }
        return arr[front];
    }

    int getRear() {
        if (isEmpty()) {
            cout << "Queue is empty.\n";
            return -1;
        }
        return arr[rear];
    }

    ~Queue() {
        delete[] arr;
    }
};

int main() {
    Queue q(5);
```

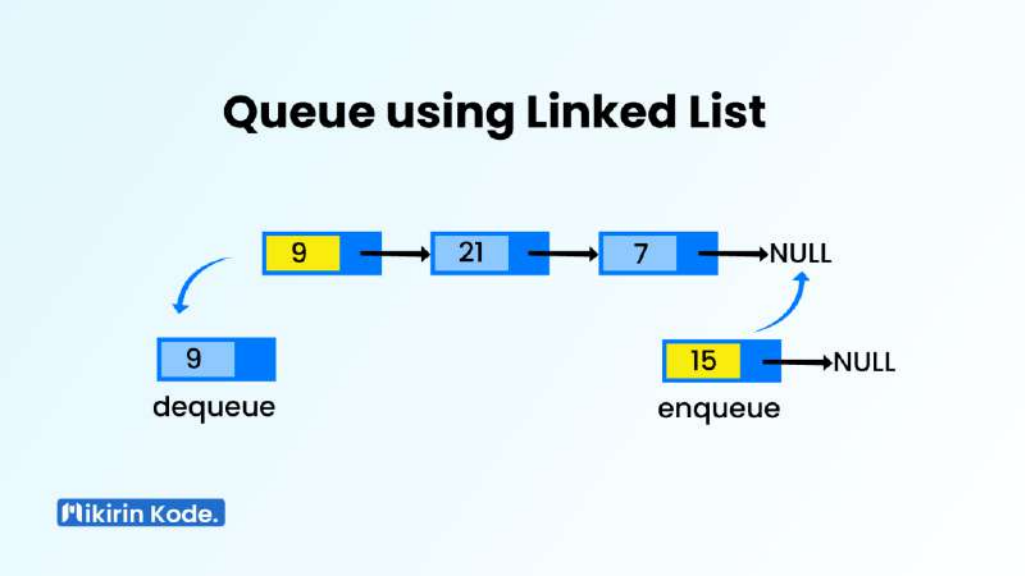
```
    q.enQueue(10);
    q.enQueue(20);
    q.deQueue();
    q.enQueue(30);

    cout << "Front element: " << q.getFront() << endl;
    cout << "Rear element: " << q.getRear() << endl;

    return 0;
}
```

Queue Using Linked Lists

- In a linked list-based implementation of a queue, we use a linked list to store the elements.
- We maintain two pointers: one for the front (where elements are dequeued) and another for the rear (where elements are enqueued).



- Here's an example of implementing a queue using a linked list in C++:
-

```
#include <iostream>
using namespace std;

struct QNode {
    int data;
    QNode* next;
    QNode(int d) : data(d), next(nullptr) {}
};

class Queue {
private:
    QNode* front;
    QNode* rear;

public:
    Queue() : front(nullptr), rear(nullptr) {}

    bool isEmpty() {
        return front == nullptr;
    }

    void enQueue(int item) {
        QNode* newNode = new QNode(item);
        if (isEmpty()) {
            front = rear = newNode;
            return;
        }
        rear->next = newNode;
        rear = newNode;
    }

    void deQueue() {
        if (isEmpty()) {
            cout << "Queue is empty. Cannot dequeue.\n";
            return;
```

```
        }
        QNode* temp = front;
        front = front->next;
        if (front == nullptr)
            rear = nullptr;
        delete temp;
    }

    int getFront() {
        if (isEmpty()) {
            cout << "Queue is empty.\n";
            return -1;
        }
        return front->data;
    }

    int getRear() {
        if (isEmpty()) {
            cout << "Queue is empty.\n";
            return -1;
        }
        return rear->data;
    }

    ~Queue() {
        while (!isEmpty())
            deQueue();
    }
};

int main() {
    Queue q;
    q.enQueue(10);
    q.enQueue(20);
    q.deQueue();
```

```
    q.enQueue(30);

    cout << "Front element: " << q.getFront() << endl;
    cout << "Rear element: " << q.getRear() << endl;

    return 0;
}
```

Remember, both array-based and linked list-based implementations have their advantages and disadvantages. Choose the one that suits your requirements! □□

4.) Applications of Queues in Various Fields

Queues, with their "First In, First Out" (FIFO) principle, are fundamental data structures with diverse applications across computer science and beyond. Here are some prominent examples:

1. Breadth-First Search (BFS):

- **Concept:** BFS explores a graph or tree level-by-level. It starts from a root node, visits all its neighbors, then proceeds to the neighbors of the neighbors, and so on.

- **How Queues Help:** A queue is used to keep track of the unvisited nodes. Nodes are added to the back of the queue (enqueue) as they are discovered, and explored one by one from the front (dequeue). This ensures that all neighbors of a node are explored before moving to the next level, achieving the BFS behavior.

2. Scheduling:

- **Context:** Operating systems, task managers, and process schedulers often rely on queues to manage tasks or processes efficiently.

- **Applications:**
  - **Job Queues:** Jobs submitted for printing, file downloads, or background tasks are often placed in a queue and processed one by one based on their arrival time or priority.
  - **CPU Scheduling:** The operating system maintains a queue of processes waiting for CPU time. Processes are dequeued and allocated CPU time based on scheduling algorithms (e.g., round-robin, priority).
  - **Disk I/O Scheduling:** Requests to read/write data from the disk are queued. The scheduler determines the order in which these requests are served to optimize disk access time.

3. Network Packet Buffering:

- **Scenario:** When data is transmitted over a network, it's broken down into smaller packets. Queues are used at both sending and receiving ends:
  - **Sending:** Packets might need to be buffered before transmission due to network congestion or flow control. A queue holds packets waiting to be sent on the network.
  - **Receiving:** Incoming packets might arrive faster than the application can process them. A queue buffers these packets until the application is ready to receive them.

4. Producer-Consumer Problem:

- **Concept:** This is a classic synchronization problem where a producer (e.g., a sensor generating data) and a consumer (e.g., a program processing the data) need to cooperate.
- **Solution using Queues:** A queue acts as a shared buffer. The producer adds items (data) to the back of the queue (enqueue). The consumer removes items (data) from the front (dequeue) for processing. Synchronization mechanisms (e.g., semaphores) are used to ensure proper access to the queue and avoid conflicts.

5. Other Applications:

- **Breadth-First Search variations:** Bi-directional BFS, iterative deepening DFS, etc.
- **Level Order Tree Traversal:** Printing the nodes of a tree level by level.
- **Implementing Undo/Redo functionality:** Storing previous states in a queue for backtracking.
- **Real-time simulations:** Queues can manage events to be processed in a specific order based on their simulated time.

Queues offer a versatile and efficient way to manage data flow and handle tasks in various scenarios. Their simplicity and FIFO nature make them a valuable tool for many computer science applications and beyond.

5.) Introduction to Deques (Double-Ended Queues)

- A deque (pronounced as “deck” or “dequeue”) is a generalized version of the queue data structure.
- It allows insertions and deletions at both ends (front and rear).

- Think of a deque as a linear collection of elements that supports operations at both ends.

Basic Operations on Deques

Below is a table showing some basic operations along with their time complexity, performed on deques:

Operation	Description	Time Complexity
`push_front()`	Inserts an element at the beginning (front).	O(1)
`push_back()`	Adds an element at the end (rear).	O(1)
`pop_front()`	Removes the first element from the deque.	O(1)
`pop_back()`	Removes the last element from the deque.	O(1)
`front()`	Gets the front element from the deque.	O(1)
`back()`	Gets the last element from the deque.	O(1)
`empty()`	Checks whether the deque is empty or not.	O(1)
`size()`	Determines the number of elements in the deque.	O(1)

Applications of Deques

Since deques support both stack and queue operations, they can be used for various purposes:

1. **Clockwise and Anticlockwise Rotations**:
    - Deques support rotations in O(1) time, which can be useful in certain applications.

2. **Problems Requiring Insertions and Deletions at Both Ends**:
    - Deques efficiently solve problems where elements need to be added to or removed from both ends.
    - Examples: Maximum of all subarrays of size k, 0-1 BFS, and finding the first circular tour that visits all petrol pumps.

Remember, deques are versatile and find applications in various algorithms and data processing tasks!
