## **LEC-7 Queue**

# **Queue: Overview**

A Queue is a fundamental data structure commonly used in computer science and programming. It follows the First-In-First-Out (FIFO) principle, meaning that the first element added to the queue is the first one to be removed. Queues have a wide range of applications, from task scheduling to breadth-first search in graphs.

# **Characteristics of a Queue**

Queues possess several key characteristics that define their behavior:

* **FIFO Principle:** The first item added to the queue is the first one to be removed.
* **Two Main Operations:** Queues support two primary operations: "enqueue" to add an element and "dequeue" to remove the front element.
* **Front and Rear:** The front points to the first element, and the rear points to the last element.

# **Queue Implementation**

Queues can be implemented using different data structures, but two common approaches are:

* **Array Implementation:** Using an array to store queue elements, with two pointers indicating the front and rear positions.
* **Linked List Implementation:** Utilizing a linked list where each node holds an element and a pointer to the next node.

# **Queue Operations**

**1. Enqueue Operation**

Adding an element to the rear of the queue:

* Allocate memory (for linked list implementation) or check for overflow (for array implementation).
* Place the new element at the rear of the queue.
* Update the queue's rear pointer.

**2. Dequeue Operation**

Removing the front element from the queue:

* Check for underflow (empty queue).
* Remove the front element.
* Update the queue's front pointer.

**3. Peek Operation**

Viewing the front element without removal:

* Return the element at the front of the queue.
* No changes to the queue's structure.

# **Common Applications**

Queues find applications in various scenarios:

* **Task Scheduling:** Queues manage tasks in an order of arrival, as in print spooling.
* **Breadth-First Search (BFS):** Queues are used to traverse levels of a graph in BFS.
* **Resource Sharing:** Queues are employed in scenarios where resources need to be shared fairly among multiple consumers.
* **Buffering:** Queues are used for buffering in various scenarios, like I/O operations.

# **Types of Queues**

Several variations of queues cater to specific needs:

* **Priority Queue:** Elements have associated priorities, and the highest-priority element is dequeued first.
* **Circular Queue:** The rear and front pointers wrap around the array, allowing for efficient space utilization.

# **Limitations**

Queues have certain limitations:

* **Fixed Size (for array implementation):** Arrays have a predetermined size, potentially leading to overflow or underflow.
* **Dynamic Memory Allocation (for linked list implementation):** Linked list nodes require memory allocation, leading to potential memory fragmentation.

# **Conclusion**

In conclusion, a Queue is a crucial data structure that adheres to the First-In-First-Out principle. Its enqueue, dequeue, and peek operations make it valuable for applications such as task scheduling, graph traversal, and resource sharing. The choice of implementation (array or linked list) and variations (priority queues, circular queues) depends on the specific requirements of the problem at hand.

//Queue

//Data Structures that stores data on first come first serve or we can say Queue in firt in out(FIFO)

#include <iostream>

using namespace std;

class queue{

private:

    int front;

    int rear;

    int maxsize;

    int currsize;

    int\* arr;

public:

    queue(int n=10){

        front=-1;

        rear=-1;

        maxsize=n;

        currsize=0;

        arr=new int [maxsize];

    }

    ~queue(){

        if(arr!=0){

            delete []arr;

            arr=0;

        }

    }

    bool isFull(){

        if(currsize==maxsize){

            return true;

        }

        return false;

    }

    bool isEmpty(){

        if(currsize==0){

            return true;

        }

        return false;

    }

    bool enqueue(int val){ //insert at end

        if(isFull()){

            return false;

        }

        else{

            rear=(rear+1)%maxsize;

            arr[rear]=val;

            currsize++;

            return true;

        }

    }

    bool dequeue(int val){ //delete from end

        if(isEmpty()){

            return false;

        }

        else{

            front=(front+1)%maxsize;

            currsize--;

            return true;

        }

    }

};

## **Queue Implementation using LinkedList**

#include <iostream>

using namespace std;

class Queue;

class Node{

    friend class Queue;

    int data;

    Node\* next;

};

class Queue{

private:

    Node\* front;

    Node\* rear;

public:

    Queue(){

        front=0;

        rear=0;

    }

    ~Queue();

    bool isEmpty();

    bool Enqueue(int val){ //insert at end

        Node\* temp=new Node;

        temp->data=val;

        temp->next=0;

        if(rear==0){

            front=temp;

            rear=temp;

            return true;

        }

        else{

            rear->next=temp;

            rear=temp;

        }

    }

    bool Dequeue(){ // delete from end

        if(isEmpty()){

            return false;

        }

        if(front==rear){

            delete rear;

            front=0;

            rear=0;

            return true;

        }

        else{

            Node\* temp =front;

            front =front->next;

            delete temp;

            temp=0;

            return true;

        }

    }

};