UIT2021805:Ananya Apte

UIT2021809:Anushka Badave

UIT2021863:Manali Selmokar

UIT2021865:Rutika Ubalekar

**DATA STRUCTURE:QUEUE**

**APPLICATION:ELEVATOR MECHANISM**

****

**What is Queue?**

A queue is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.

We define a queue to be a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end. The element which is first pushed into the order, the operation is first performed on that.

**Characteristics of Queue:**

Queue can handle multiple data.

We can access both ends.

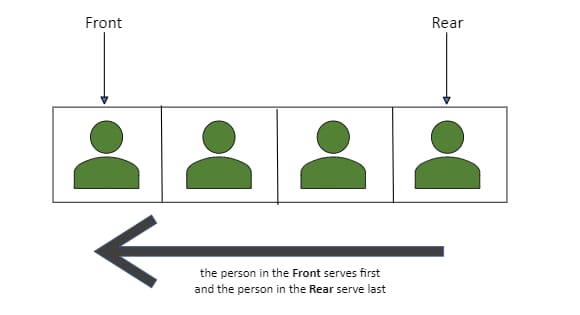
They are fast and flexible.

**FIFO Principle of Queue:**

A Queue is like a line waiting to purchase tickets, where the first person in line is the first person served. (i.e. First come first serve).

Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the front of the queue(sometimes, head of the queue), similarly, the position of the last entry in the queue, that is, the one most recently added, is called the rear (or the tail) of the queue.

See the below figure.



**Queue Representation:**

Like stacks, Queues can also be represented in an array: In this representation, the Queue is implemented using the array. Variables used in this case are

**Queue:** the name of the array storing queue elements.

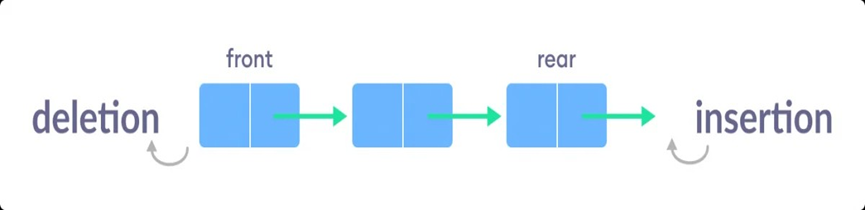
**Front**: the index where the first element is stored in the array representing the queue.

**Rear**: the index where the last element is stored in an array representing the queue

**Types of Queue:**

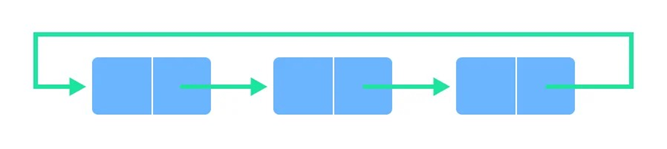
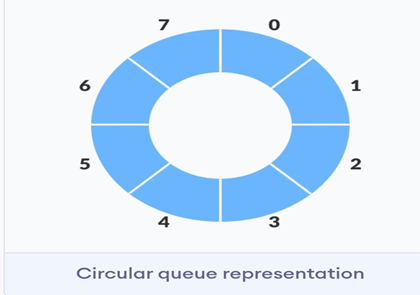
● **Simple Queue**

In a simple queue, insertion takes place at the rear and removal occurs at the front. It strictly follows the FIFO (First in First out) rule.A simple queue is the most basic queue. In this queue, the enqueue operation takes place at the rear, while the dequeue operation takes place at the front.Its applications are process scheduling, disk scheduling, memory management, IO buffer, pipes, call center phone systems, and interrupt handling.



● **Circular Queue**

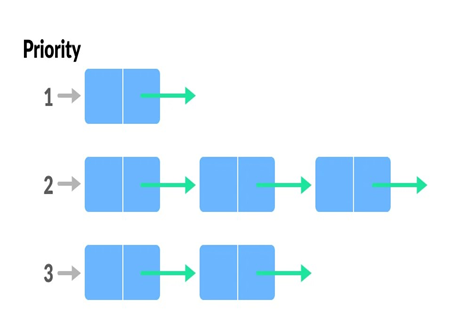
In a circular queue, the last element points to the first element making a circular link.A Circular Queue is a special version of queue where the last element of the queue is connected to the first element of the queue forming a circle.A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a Ring Buffer.



● **Priority Queue**

A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.



**QUEUE Methods:**

**add()** - Inserts the specified element into the queue. If the task is successful, add() returns true, if not it throws an exception.

**offer()** - Inserts the specified element into the queue. If the task is successful, offer() returns true, if not it returns false.

**element()** - Returns the head of the queue. Throws an exception if the queue is empty. peek() - Returns the head of the queue. Returns null if the queue is empty.

**remove()** - Returns and removes the head of the queue. Throws an exception if the queue is empty.

**poll()** - Returns and removes the head of the queue. Returns null if the queue is empty.

**Elevator mechanism:**

****

**How is the lift assigned to the user?**

When a user presses his desired floor on the keypad, the system goes through the following steps:

1) It checks if any lift is on that floor, if yes, it assigns the lift.

2) If no, it checks whether the user wants to go to an upper floor or lower floor from his current floor.

3) It then assigns the lift which is going up and nearer to that floor.

The main constraint in this algorithm is that a lift can have a maximum on 4 stops. So when it checks a lift, it checks whether the lift has reached it’s maximum stops or no.

**What’s the system and how it works?**

Each lift is connected to an Arduino in the master-slave configuration. Each lift is a slave to a master Arduino which is responsible to assign the lift to the user.

Each lift maintains a queue(data structure). The number of elements in the queue is the maximum number of stops a lift can have.

The first element in the queue is its current floor. The subsequent elements are its destinations in order.

The master checks each queue and gets where all the lifts are from the first element of their queues.

The lift goes upwards when the difference between the current element and next element is positive and vice-versa.

**Why only Priority queue?**

Using Stack: Last in, first-out . Serve those with the shortest request first. Some people never get off the elevator.

Using queue: First in, first-out. Serve those who have been on the elevator the longest. People ride the elevator up and down, potentially passing up their floor many times.

Using priority queue: Serve riders based on priority. High priority means direct travel to your floor.

**Class Diagram:**

**ElevatorManager**

[This is the main elevator program which will manage n elevators in the building]

**Members:**

1.List of Elevator

2.Queue of floor request (This maintains request for both directions. One improvement could be to keep two queues, one for each direction but it would increase complexity)

3.Minimum floor

4.max floor

**Operations:**

1.delgate()

2.halt() (set whole elevator system in maintenance mode or halt operation)

**Elevator**

[Represents individual elevators. There could be n elevators in a building]

**Members:**

Queue of Floor (this needs to be sorted so may be a PriorityQueue could be used)

Direction :

Enum (Enum of direction - up, down, wait, idle, maintenance)

CurrentFloor : Floor

**Operations:**

operate()

moveUp()

moveDown()

openDoor()

closeDoor()

callEmergencyLine()

getDirection()

getCurrentFloor()

setInMaintenanceMode()

**Floor**

[Represents individual floors]

**Members:**

Enum of Floors

class Request

{

currentFloor

destinationFloor

Direction [Up, Down]

**Operation:**

goUp()

goDown

**Pseudocode:**

class Floor {

goUp() {

ElevatorManager.queue.offer(new Request(currentFloor, destinationFloor, up));

//offer() - Inserts the specified element into the queue.

}

goDown() {

ElevatorManager.queue.offer(new Request(currentFloor, destinationFloor, down))}

}

class ElevatorManager {

delegate() {

Elevator idleElevator; // track idle elevator

Elevator elevatorMovingInSameDirection;

// elevator moving in same direction as next request in main elevator manager queue

while(!stop()) { //keep delegating until powered down or whole system is halted through main

controls

if(queue.peek() != null) { //peek() - Returns the head of the queue. Returns null if the queue is

empty.

Request req = queue.peek();

boolean startAgain = false;

// flag to start from beginning if the request is already pushed

----

// first find if there is an elevator at current floor going in same direction as current request in

queue

if(req.currentFloor == elevator.currentFloor && req.direction == elevator.direction) {

elevator.queue.offer(req.destinationFloor);

queue.poll(); // remove this request from Elevator Manager queue

//poll() - Returns and removes the head of the queue. Returns null if the queue is empty.

startAgain = true;

break; }

// check if this elevator is idle

if(elevator.direction == "idle")) {

idleElevator = elevator;}

----

// check if this elevator is moving in desired direction and elevator's current floor is behind

desired floor in queue

if(elevator.direction == req.direction) {

// Make sure elevators moving in same direction should also be behind the floor where request is

made

if(req.direction == "Up" && req.currentFloor - elevator.currentFloor > 0) {

elevatorMovingInSameDirection = elevator;

}

// Make sure elevators moving in same direction should also be behind the floor where request is

made

if(req.direction == "Down" && req.currentFloor - elevator.currentFloor < 0) {

elevatorMovingInSameDirection = elevator;}}}}}

public class Elevator {

moveUp() {

this.currentFloor += 1; }

moveDown() {

this.currentFloor -= 1; }

----

operate() {

while(queue.peek() != null) {

//peek() - Returns the head of the queue. Returns null if the queue is empty. Floor nextFloorInQueue = queue.peek();

while(this.currentFloor != nextFloorInQueue.request.destinationFloor) {

if(this.direction == "Up") {

moveUp(); }

else if(this.direction == "down") {

moveDown(); } }

queue.poll(); // remove the request from queue

open(); //open door

Direction backUpDirection = this.direction;

//back up elevators direction to retrieve it later once dooor closes

this.direction = "idle"; // set state to idle to let elevatorManager know that requests at current floor could be offered to this elevator queue

Thread.sleep(10000); // sleep for 10 seconds so that people can leave elevator

close(); // once people are out close door to move to next floor in queue

this.direction = backUpDirection; }

this.direction = "idle"; // once queue is empty set the direction to idle } }

**Applications using Queue:**

1. In a computer system, there may be queues of tasks waiting for the printer, for access to disk storage, or even in a time-sharing system, for use of the CPU.

2.In a network, a queue is used in devices such as a router/switch and mail queue.

3.Managing requests on a single shared resource such as CPU scheduling and disk scheduling.

4.Handling hardware or real-time systems interrupts.

5.Handling website traffic.

6.Routers and switches in networking.

7.Maintaining the playlist in media players.

**References:**

1.javatpoint:<https://www.javatpoint.com/data-structure-queue>

2.geeksforgeeks:<https://www.geeksforgeeks.org/queue-data-structure/?ref=lbp>

3.stackoverflow:<https://stackoverflow.com/questions/12009556/datastructure-for-elevator-mechanism>

4.codingninjas:<https://www.codingninjas.com/codestudio/library/queue>

5.canva(for poster)

**Contributions:**

**Info collection:Ananya,Rutika**

**Poster Making:Manali,Ananya**

**Pseudocode:Manali**

**Report:Anushka,Rutika**