Data Structures [R1UC308B]

Module-VII: Queue

Dr. Subhash Chandra Gupta

School of Computer Science and Engineering
Plat No 2, Sector 17A, Yamuna Expressway
Greater Noida, Uttar Pradesh - 203201

July 31, 2025

Contents

Introduction to Queue

- A Queue is a fundamental concept in computer science used for storing and managing data in a specific order.
- It follows the principle of "First in, First out" (FIFO), where the first element added to the queue is the first one to be removed.
- Queues are commonly used in various algorithms and applications for their simplicity and efficiency in managing data flow.

Figure: Structure of Queue

Operations on Queue

- Create - Creating the queue with initial values.
- Add - enqueue() – Insertion of elements at rear to the queue.
- Delete - dequeue() – Removal of elements at front from the queue.
- Full - isFull() – Validates if the queue is full.
- Empty - isEmpty() – Checks if the queue is empty.
- peek() or front()- Acquires the data element available at the front node of the queue without deleting it.
- rear() – This operation returns the element at the rear end without removing it.
- size(): This operation returns the size of the queue i.e. the total number of elements it contains at present.

Implementation of queues

- Array
- Linked List

Array implementation of queues

```
//Queue using Array

class Queue {
int queue[];
int front, rear, capacity;

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

    // insert an element into the queue
    void enQueue(int item)  {
        // check if the queue is full
        if (capacity-1 == rear) {
            System.out.println("Queue is full");
```

```
            return;
        }
        // insert element at the rear
        else {
            queue[++rear] = item;
            if(front==-1) front=0;
        }
        return;
    }

    //remove an element from the queue
    int deQueue()  {
     // check if queue is empty
        if (front==-1 || rear<front) {
            System.out.println("Queue is empty");
            return Integer.MIN_VALUE;
        }
```

```
        else {
          int temp=queue[front++];
          System.out.printf("Item "+temp+" is deQueued\n");
          return temp;
         }

    }

    public boolean isEmpty()  {
return (front==-1 || rear<front);
}

    public boolean isFull()  {
return (rear==capacity-1);
}

    // print queue elements
```

```
void queueDisplay()
{
    int i;
    if (front == -1 || rear<front) {
        System.out.println("Queue is Empty");
        return;
    }

    // traverse front to rear and print elements
    System.out.printf("Elements of the Queue are: ");
    for (i = front; i < rear; i++) {
        System.out.printf("%d , ", queue[i]);
    }
    System.out.printf(" %d\n", queue[i]);
    return;
}
```

```
// print front of queue
void queueFront()
{
    if (front == -1) {
        System.out.println("Queue is Empty");
        return;
    }
    System.out.printf("Front Element of the queue: %d\n
    return;
}
void queueRear()
{
 if (front == -1) {
        System.out.println("Queue is Empty");
        return;
    }
    System.out.printf("Rear Element of the queue: %d\n'
```

```
        return;
    }
    int queueSize()
    {
     int size=0;
        if (front == -1 || rear<front) return size;
        if(front<=rear)
         size=rear-front+1;
        return size;
    }
}

public class QueueArray  {
    public static void main(String[] args) {
        // Create a queue of capacity 4
        Queue q = new Queue(4);
```

```
System.out.println("Initial Queue(capacity=4):");
// print Queue elements
System.out.println("Display():");
q.queueDisplay();
System.out.printf("Size():%d\n",q.queueSize());

// inserting elements in the queue
System.out.println("enQueue(10):");
q.enQueue(10);
q.queueDisplay();
System.out.printf("Size():%d\n",q.queueSize());

System.out.println("enQueue(30):");
q.enQueue(30);
q.queueDisplay();
System.out.printf("Size():%d\n",q.queueSize());
```

```
System.out.println("enQueue(50):");
q.enQueue(50);
q.queueDisplay();
System.out.printf("Size():%d\n",q.queueSize());

System.out.println("enQueue(70):");
q.enQueue(70);
q.queueDisplay();
System.out.printf("Size():%d\n",q.queueSize());

// insert element in the queue
System.out.println("enQueue(90):");
q.enQueue(90);
q.queueDisplay();
System.out.printf("Size():%d\n",q.queueSize());

System.out.println("deQueue():");
```

```
q.deQueue();
q.queueDisplay();
System.out.printf("Size():%d\n",q.queueSize());

System.out.println("deQueue():");
q.deQueue();
q.queueDisplay();
System.out.printf("Size():%d\n",q.queueSize());

System.out.println("enQueue(100):");
q.enQueue(100);
q.queueDisplay();
System.out.printf("Size():%d\n",q.queueSize());

System.out.println("enQueue(110):");
q.enQueue(110);
q.queueDisplay();
```

```
        System.out.printf("Size():%d\n",q.queueSize());

        System.out.println("enQueue(120):");
        q.enQueue(120);
        q.queueDisplay();
        System.out.printf("Size():%d\n",q.queueSize());

        System.out.println("Front():");
        q.queueFront();
        System.out.println("Rear():");
        q.queueRear();
    }
}
```

Linked implementation of queues

```
//Queue using Linked List

class QueueLL {
private Node front, rear;
private int queueSize; // queue size

//linked list node
private class Node {
int data;
Node next;
}

//default constructor - initially front & rear are null; si
public QueueLL()  {
front = null;
rear = null;
queueSize = 0;
```

```
}
//check if the queue is empty
public boolean isEmpty()  {
return (queueSize == 0);
}

//Remove item from the front of the queue.
public int dequeue() {
if (isEmpty()) {
System.out.println("\nQueue is empty");
}
int data = front.data;
front = front.next;
queueSize--;
if (isEmpty()) {//queueSize is 0 after dequeue.
rear = null;
}
```

```
System.out.println("Element " + data+ " removed from the qu
return data;
}

//Add data at the rear of the queue.
public void enqueue(int data) {
Node New_Node=new Node();
if(New_Node==null) {
System.out.println("\nQueue is full");
}
New_Node.data=data;
New_Node.next=null;

if (isEmpty()){
front = rear=New_Node;
}
```

```
else  {
rear.next=New_Node;
rear=New_Node;
}
queueSize++;
System.out.println("Element " + data+ " added to the queue'
}
//print front and rear of the queue
public void print_frontRear() {
if(front!=null)
System.out.println("Front of the queue:" + front.data
+ "\nRear of the queue:" + rear.data);
}
}

class QueueLink{
public static void main(String a[]){
```

```
QueueLL queue = new QueueLL();
queue.enqueue(6);
queue.enqueue(3);
queue.print_frontRear();
queue.enqueue(12);
queue.enqueue(24);
queue.dequeue();
queue.dequeue();
queue.enqueue(9);
queue.print_frontRear();
}
}
```

Circular queues

- A Circular Queue is an extended version of a normal queue where the last element of the queue is connected to the first element of the queue forming a circle.
- The operations are performed based on FIFO (First In First Out) principle. It is also called 'Ring Buffer'.
- In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.

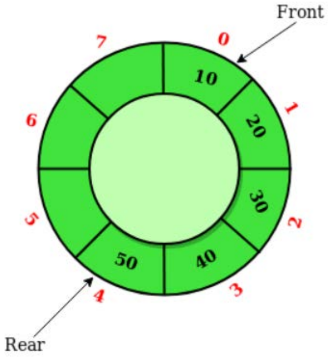

Figure: Circular Queue

```
//Circular Queue using Array

class QueueA

{
int queue[];
int front;
int rear;
int capacity;

    QueueA(int size) {
        front = rear = -1;
        capacity = size;
        queue = new int[capacity];
    }

    // insert an element into the queue
```

```
void enQueue(int item)  {
    // check if the queue is full
    if (front == ((rear+1)%capacity)) {
        System.out.println("Queue is full");
        return;
    }
    // insert element at the rear
    else {
     rear= (rear==capacity-1?0:++rear);
     //rear=((rear+1)%capacity);
        queue[rear] = item;
        if(front==-1) front=0;
        queueDisplay();
    }
    return;
}
```

```
//remove an element from the queue
int deQueue()  {
 // check if queue is empty
    if (front ==-1) {
        System.out.println("Queue is empty");
        return -1;
    }
   else {
     int temp=queue[front];
     if(front==rear)
     front=rear=-1;
     else
     front=(front==capacity-1?0:++front);
     //front=((front+1)%capacity);
     System.out.printf("Item "+temp+" is deQueued\n");
     queueDisplay();
     return temp;
```

```
    }

}

// print queue elements
void queueDisplay()
{
    int i;
    if (front == -1) {
        System.out.println("Queue is Empty");
        return;
    }

    // traverse front to rear and print elements
    System.out.println("Queue is:");
    if(front<=rear)
     for (i = front; i < rear; i++)
```

```
        System.out.printf(" %d , ", queue[i]);
       else {
        for (i = front; i < rear+capacity; i++)
        System.out.printf(" %d , ", queue[i%capacity]);
        //for (i = 0; i < rear; i++)
        //System.out.printf(" %d , ", queue[i]);
       }
       System.out.printf(" %d \n", queue[rear]);

       return;
   }

   // print front of queue
   void queueFront()
   {
       if (front == -1) {
           System.out.println("Queue is Empty");
```

```
        return;
    }
    System.out.printf("Front Element of the queue: %d\n
    return;
}

void queueRear()
{
 if (front == -1) {
        System.out.println("Queue is Empty");
        return;
    }
    System.out.printf("Rear Element of the queue: %d\n"
    return;
}

int queueSize()
```

```
    {
     int size=0;
        if (front == -1) return size;

        if(front<=rear)
         size=rear-front+1;
        else
         size=capacity-(front-rear-1);
        return size;
    }
}

public class QueueC  {
    public static void main(String[] args) {
        // Create a queue of capacity 4
        QueueA q = new QueueA(4);
```

```
System.out.println("Initial Queue(capacity=4):");
// print Queue elements
System.out.println("Display():");
q.queueDisplay();
System.out.printf("Size():%d\n",q.queueSize());

// inserting elements in the queue
System.out.println("enQueue(10):");
q.enQueue(10);
System.out.printf("Size():%d\n",q.queueSize());

System.out.println("enQueue(30):");
q.enQueue(30);
System.out.printf("Size():%d\n",q.queueSize());

System.out.println("enQueue(50):");
q.enQueue(50);
```

```
System.out.printf("Size():%d\n",q.queueSize());

System.out.println("enQueue(70):");
q.enQueue(70);
System.out.printf("Size():%d\n",q.queueSize());

// insert element in the queue
System.out.println("enQueue(90):");
q.enQueue(90);
System.out.printf("Size():%d\n",q.queueSize());

System.out.println("deQueue():");
q.deQueue();
System.out.printf("Size():%d\n",q.queueSize());

System.out.println("deQueue():");
q.deQueue();
```

```
System.out.printf("Size():%d\n",q.queueSize());

System.out.println("enQueue(100):");
q.enQueue(100);
System.out.printf("Size():%d\n",q.queueSize());

System.out.println("enQueue(110):");
q.enQueue(110);
System.out.printf("Size():%d\n",q.queueSize());

System.out.println("enQueue(120):");
q.enQueue(120);
System.out.printf("Size():%d\n",q.queueSize());

System.out.println("Front():");
q.queueFront();
System.out.println("Rear():");
```

```
        q.queueRear();
    }
}
```

```
//Circular Queue using Linked List

class QueueCLL {
private Node front, rear;
private int queueSize; // queue size

//linked list node
private class Node {
int data;
Node next;
}

//default constructor - initially front & rear are null;
//size=0; queue is empty
public QueueCLL()  {
front = null;
rear = null;
```

```
queueSize = 0;
}
//check if the queue is empty
public boolean isEmpty()  {
return (queueSize == 0);
}

//Add data at the rear of the queue.
@SuppressWarnings("unused")
public void enqueue(int data) {
Node New_Node=new Node();
if(New_Node==null) {
System.out.println("Queue is full");
}
New_Node.data=data;
if (isEmpty()){
front = rear=New_Node;
```

```
rear.next=front;
}
else  {
New_Node.next=front;
rear.next=New_Node;
rear=New_Node;
}
queueSize++;
System.out.println("Element " + rear.data+ " added to the q
}

//Remove item from the front of the queue.
public int dequeue() {
if (isEmpty()) {
System.out.println("Queue is empty");
}
int data = front.data;
```

```
front = front.next;
rear.next=front;
queueSize--;
if (isEmpty()) {//queueSize is 0 after dequeue.
front=rear = null;
}

System.out.println("Element " + data+ " removed from the qu
return data;
}

//print front and rear of the queue
public void print_frontRear() {
if(front!=null)
System.out.println("Front of the queue:" + front.data
+ "\nRear of the queue:" + rear.data);
}
```

```
void queueDisplay()
    {
        Node temp;
        if (front == null) {
            System.out.println("Queue is Empty");
            return;
        }

        // Traverse front to rear and print elements
        System.out.printf("Elements of the Queue: ");
        for (temp = front; temp !=  rear; temp=temp.next) {
            System.out.printf("%d , ", temp.data);
        }
        System.out.printf(" %d\n", rear.data);
        return;
    }
}
```

```
class QueueCirLL{
public static void main(String a[]){
QueueCLL q = new QueueCLL();
System.out.println("Display():");
q.queueDisplay();
System.out.println("enqueue(6):");
q.enqueue(6);
q.queueDisplay();
System.out.println("enqueue(3):");
q.enqueue(3);
q.queueDisplay();
System.out.println("enqueue(12):");
q.enqueue(12);
q.queueDisplay();
System.out.println("enqueue(24):");
q.enqueue(24);
```

```
q.queueDisplay();
System.out.println("dequeue():");
q.dequeue();
q.queueDisplay();
System.out.println("dequeue():");
q.dequeue();
q.queueDisplay();
System.out.println("enqueue(9):");
q.enqueue(9);
q.queueDisplay();
System.out.println("frontRear():");
q.print_frontRear();
}
}
```

Double Ended queue

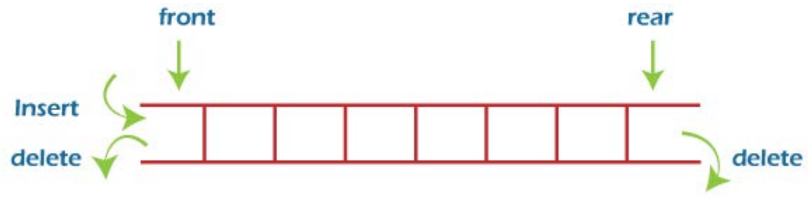

Figure: Deque

- Deque or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends.

- push-front() - Inserts the element at the beginning.
- push-back() - Adds element at the end.
- pop-front() - Removes the first element from the deque.
- pop-back() - Removes the last element from the deque.
- front() - Gets the front element from the deque.
- back() - Gets the last element from the deque.

```
//De-queue using circular array

/*Operations on Deque:
void Push_Front(int key);
void Push_Rear(int key);
void Pop_Front();
void Pop_Rear();
bool isFull();
bool isEmpty();
int getFront();
int getRear();*/
class DequeCirArr {
static int queue[];
static int front;
static int rear;
static int capacity;
```

```
public DequeCirArr(int size)
{
front = -1;
rear = -1;
capacity = size;
queue = new int[capacity];
}

// Checks whether Deque is full or not.
boolean isFull()
{
return ((front == 0 && rear == capacity - 1)
|| front == rear + 1);
}

// Checks whether Deque is empty or not.
boolean isEmpty() { return (front == -1); }
```

```
// Inserts an element at front
void Push_Front(int key)
{
// check whether Deque is full or not
if (isFull()) {
System.out.println("Overflow");
return;
}

// If queue is initially empty
if (front == -1) {
front = 0;
rear = 0;
}
// front is at first position of queue
else if (front == 0)
```

```
front = capacity - 1;
else // decrement front end by '1'
front--;
// insert current element into Deque
queue[front] = key;
}

// function to insert element at rear end of Deque.
void Push_Rear(int key)
{
if (isFull()) {
System.out.println(" Overflow ");
return;
}

// If queue is initially empty
if (front == -1) {
```

```
front = 0;
rear = 0;
}
// rear is at last position of queue
else if (rear == capacity - 1)
rear = 0;
// increment rear end by '1'
else
rear++;
// insert current element into Deque
queue[rear] = key;
}

// Deletes element at front end of Deque
void Pop_Front()
{
// check whether Deque is empty or not
```

```
if (isEmpty()) {
System.out.println("Queue Underflow\n");
return;
}

// Deque has only one element
if (front == rear) {
front = -1;
rear = -1;
}
else
// back to initial position
if (front == capacity - 1)
front = 0;
else // increment front by '1' to remove current
// front value from Deque
front++;
```

```
}

// Delete element at rear end of Deque
void Pop_Rear()
{
if (isEmpty()) {
System.out.println(" Underflow");
return;
}

// Deque has only one element
if (front == rear) {
front = -1;
rear = -1;
}
else if (rear == 0)
rear = capacity - 1;
```

```
else
rear--;
}

// Returns front element of Deque
int getFront()
{
// check whether Deque is empty or not
if (isEmpty()) {
System.out.println(" Underflow");
return -1;
}
return queue[front];
}

// function return rear element of Deque
int getRear()
```

```
{
// check whether Deque is empty or not
if (isEmpty()) {
System.out.println(" Underflow");
return -1;
}
return queue[rear];
}

// print queue elements
    static void DequeDisplay()
    {
        int i;
        if (front == -1) {
            System.out.println("Queue is Empty");
            return;
        }
```

```
//traverse front to rear and print elements
System.out.println("Queue is:");
if(front<=rear) {
 for (i = front; i < rear; i++)
 System.out.printf(" %d , ", queue[i]);
}
else {
 for (i = front; i < rear+capacity; i++)
 System.out.printf(" %d , ", queue[i%capacity]);
 //for (i = 0; i < rear; i++)
 //System.out.printf(" %d , ", queue[i]);
}
System.out.printf(" %d \n", queue[rear]);

return;
}
```

```
public static void main(String[] args)
{

DequeCirArr dq = new DequeCirArr(5);

// Function calls
System.out.println("Initial Queue(capacity=5):");
DequeDisplay();
System.out.println("Insert element at rear: 5 ");
dq.Push_Rear(5);
DequeDisplay();

System.out.println("Insert element at rear: 10 ");
dq.Push_Rear(10);
DequeDisplay();
```

```
dq.Pop_Rear();
System.out.println("After delete rear element: ");
DequeDisplay();

System.out.println("Insert element at front: 15 ");
dq.Push_Front(15);
DequeDisplay();

System.out.println("Insert element at front: 20 ");
dq.Push_Front(20);
DequeDisplay();

System.out.println("Insert element at rear: 30 ");
dq.Push_Rear(30);
DequeDisplay();
```

```
System.out.println("Insert element at rear: 40 ");
dq.Push_Rear(40);
DequeDisplay();

System.out.println("Insert element at rear: 50 ");
dq.Push_Rear(50);
DequeDisplay();

System.out.println("Insert element at front: 60 ");
dq.Push_Front(60);
DequeDisplay();

dq.Pop_Front();
System.out.println("After delete front element:");
DequeDisplay();

System.out.println("get front element: "+ dq.getFront());
```

```
System.out.println("get rear element : "+ dq.getRear());
}
}
```

Priority Queue

- A priority queue is a type of queue that arranges elements based on their priority values.
- Elements with higher priority values are typically retrieved or removed before elements with lower priority values.
- Each element has a priority value associated with it.
- When we add an item, it is inserted in a position based on its priority value.
- There are several ways to implement a priority queue, including using an array, linked list, heap, or binary search tree.
- Binary heap being the most common method to implement.

- The reason for using Binary Heap is simple, in binary heaps, we have easy access to the min (in min heap) or max (in max heap) and binary heap being a complete binary tree are easily implemented using arrays.
- Since we use arrays, we have cache friendliness advantage also.
- Priority queues are often used in real-time systems, where the order in which elements are processed is not simply based on the fact who came first (or inserted first), but based on priority.
- Priority Queue is used in algorithms such as Dijkstra's algorithm, Prim's algorithm, Kruskal's algorithm and Huffnam Coding.

Thank you

Please send your feedback or any queries to
subhash.chandra@galgotiasuniversity.edu.in