



SNS COLLEGE OF TECHNOLOGY

Coimbatore-35
An Autonomous Institution



Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A+' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF AIML

19ITT101-PROGRAMMING IN C AND DATA STRUCTURES

I YEAR - II SEM

UNIT 4 – STACK AND QUEUE

TOPIC 2 – Queue ADT



INTRODUCTION

- Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends.
- One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.
- A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

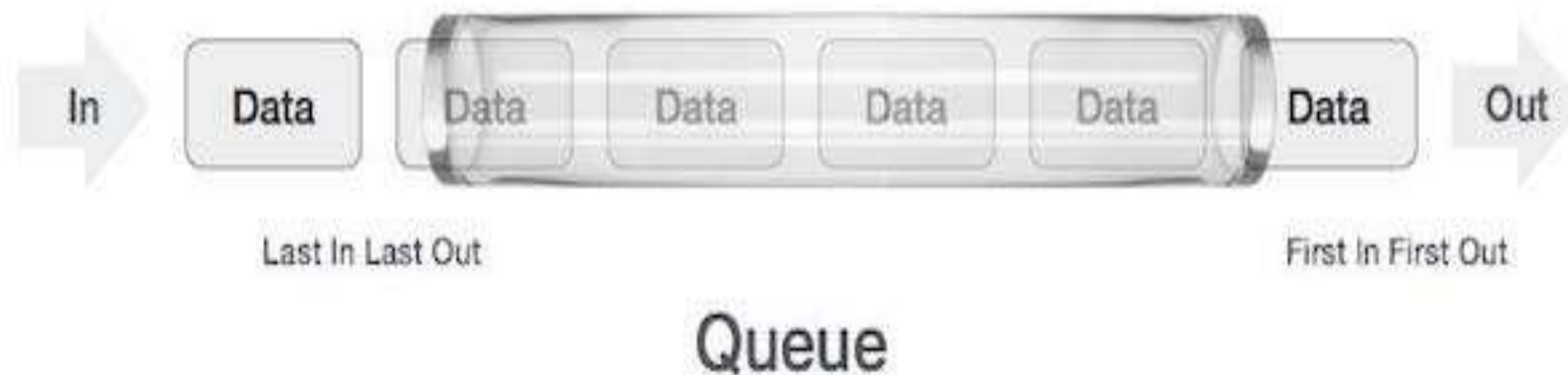




Queue Representation



- This feature makes it FIFO data structure. FIFO stands for First-in-first-out.
- The following diagram given below tries to explain queue representation as data structure.





Operation of Queue



- **enqueue()** – add (store) an item to the queue.
- **dequeue()** – remove (access) an item from the queue.
- **peek()** – Gets the element at the front of the queue without removing it.
- **isFull()** – check if Queue is full.
- **isEmpty()** – check if Queue is empty.



peek()



- **peek()** – Gets the element at the front of the queue without removing it.

```
int peek() {  
    return queue[front];  
}
```



isfull()



➤ **isFull()** – check if Queue is full.

```
bool isfull() {  
    if(rear == MAXSIZE - 1)  
        return true;  
    else  
        return false;  
}
```



isEmpty()



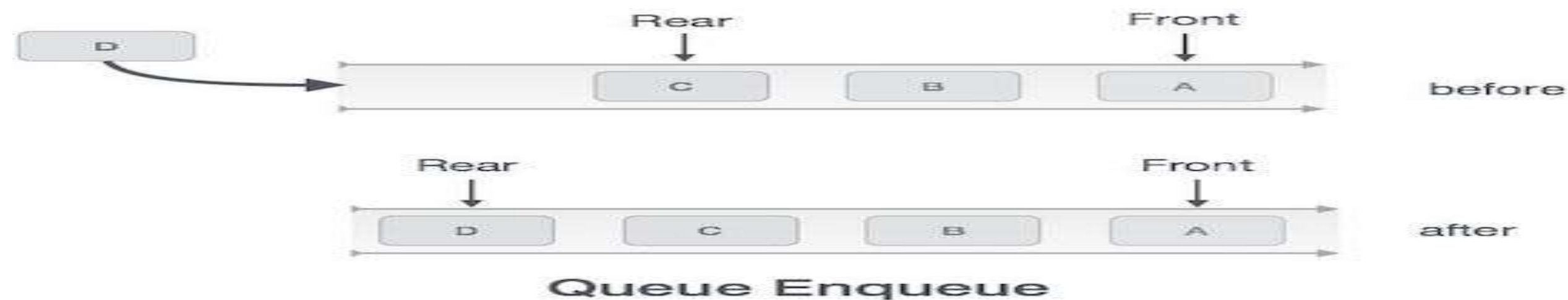
- **isEmpty()** – check if Queue is empty

```
bool isEmpty() {  
    if(front < 0 || front > rear)  
        return true;  
    else  
        return false;  
}
```




enqueue()

- **enqueue()** – add (store) an item to the queue.
- Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.
 - ❖ **Step 1** – Check if the queue is full.
 - ❖ **Step 2** – If the queue is full, produce overflow error and exit.
 - ❖ **Step 3** – If the queue is not full, increment rear pointer to point the next empty space.
 - ❖ **Step 4** – Add data element to the queue location, where the rear is pointing.
 - ❖ **Step 5** – Returns success.





enqueue()

```
int enqueue(int data)
    if(isfull())
        return 0;

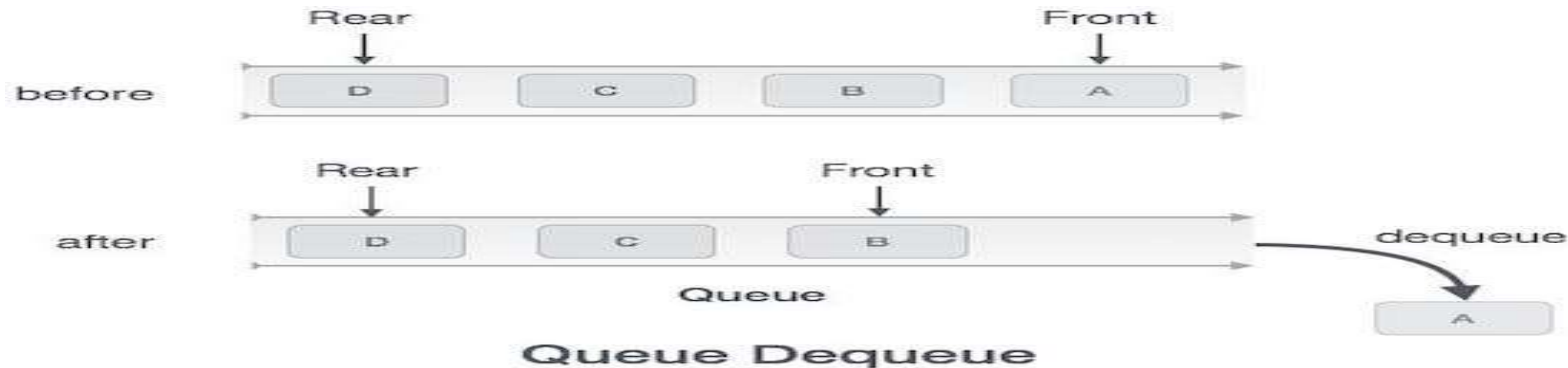
    rear = rear + 1;
    queue[rear] = data;

    return 1;
end procedure
```



dequeue()

- **dequeue()** – remove (access) an item from the queue.
- ❖ **Step 1** – Check if the queue is empty.
- ❖ **Step 2** – If the queue is empty, produce underflow error and exit.
- ❖ **Step 3** – If the queue is not empty, access the data where front is pointing.
- ❖ **Step 4** – Increment front pointer to point to the next available data element.
- ❖ **Step 5** – Return success.





dequeue()



```
int dequeue() {  
    if(isempty())  
        return 0;  
  
    int data = queue[front];  
    front = front + 1;  
  
    return data;  
}
```

https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.htm