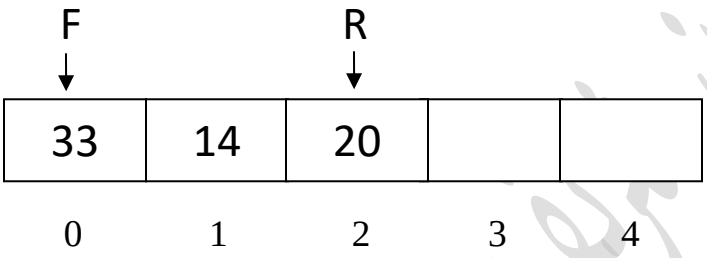
Lecture 7 The Queue

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



An example of Queue data structure

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

Operations Performed on the Queue

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue

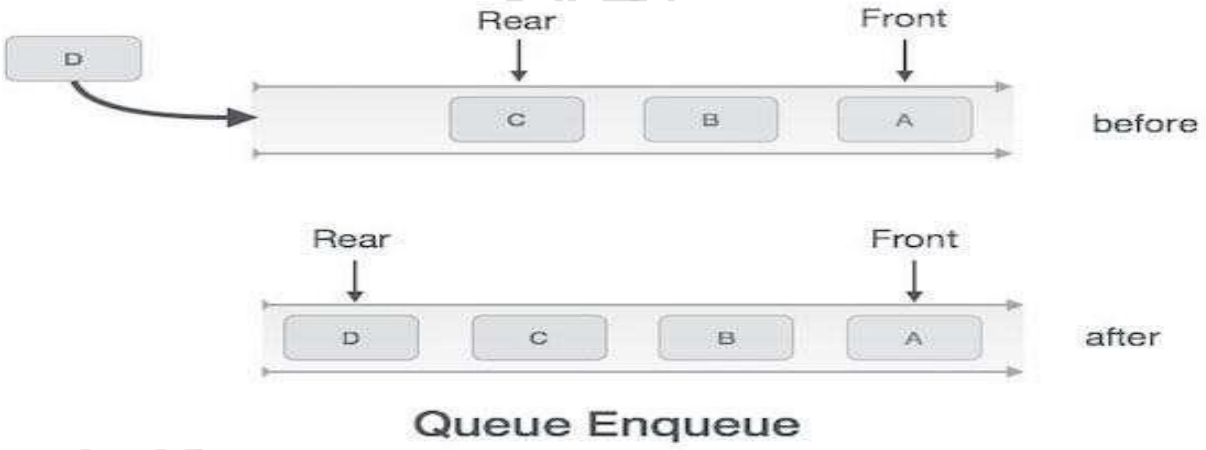
In queue, we always dequeue (or access) data, pointed by **front** pointer and while

enqueing (or storing) data in the queue we take help of **rear** pointer.
Let's first learn about supportive functions of a queue

Enqueue operation:

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are
comparatively difficult to implement than that of stacks.
The following steps should be taken to enqueue (insert) data into a queue −
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** − Return success



Queue Enqueue

Algorithm for enqueue Operation

```
 procedure enqueue(data)
if queue is full
        return overflow
endif
rear ← rear + 1
queue[rear] ← data
```

return true
end procedure

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to
perform **dequeue** operation −

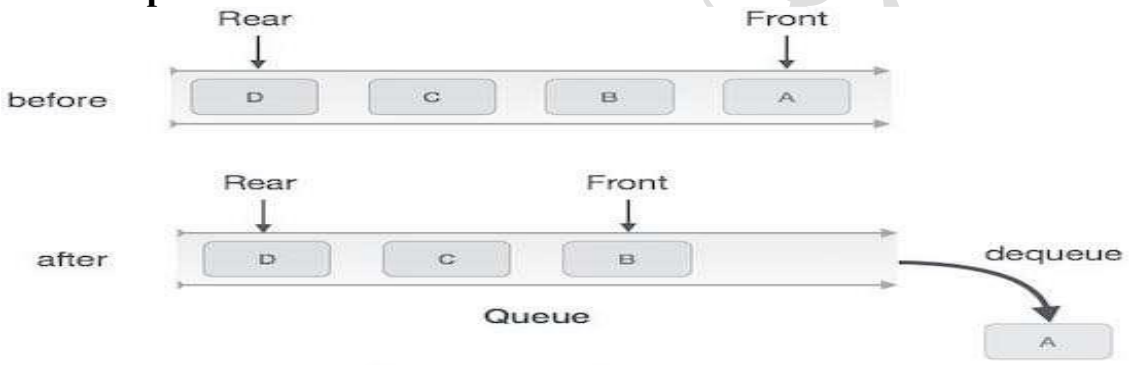
□□**Step 1** − Check if the queue is empty.

□□**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

- **Step 5** − Return success



Queue Dequeue

Algorithm for dequeue Operation

procedure dequeue
 if queue is empty
  return underflow
 end if

 data = queue[front]
 front ← front + 1
 return true
end procedure

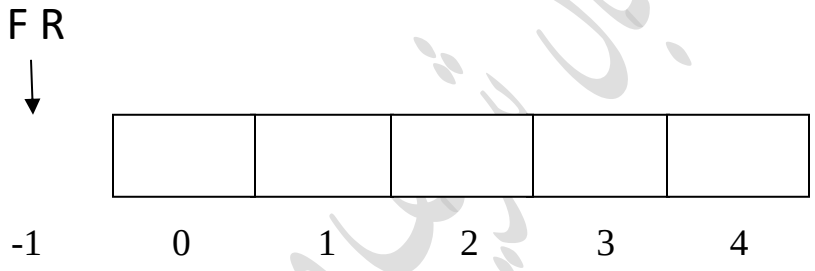
Example:

Draw figures to represent the contents of the queue and the location of front and rear pointers after performing the following add and delete operations:

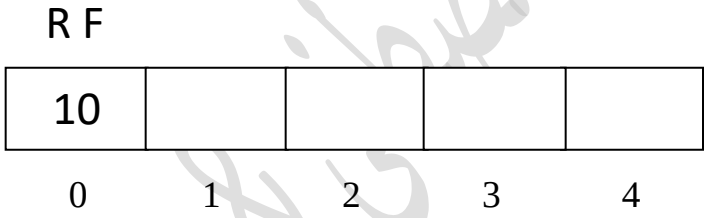
Queue Initialization, ADD(10), ADD(20), DELETE, ADD(30), ADD(40), DELETE, ADD(50), DELETE, DELETE, DELETE.

Solution:

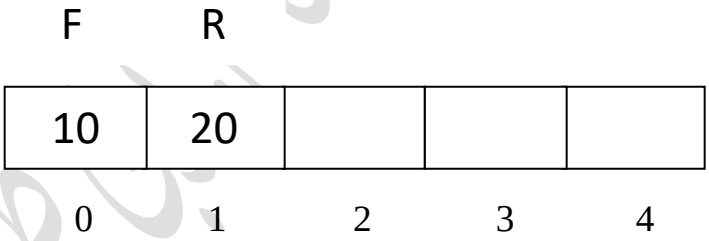
Queue Initialization:



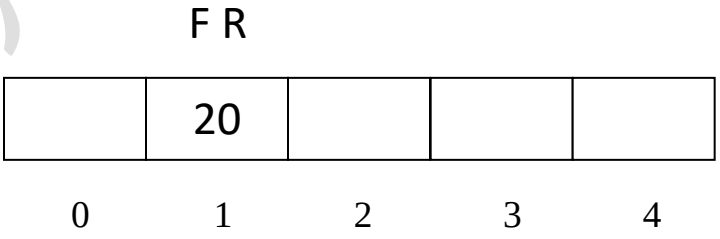
ADD(10):



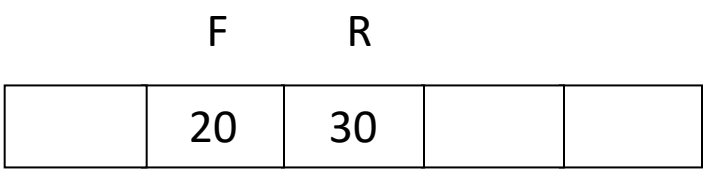
ADD(20):

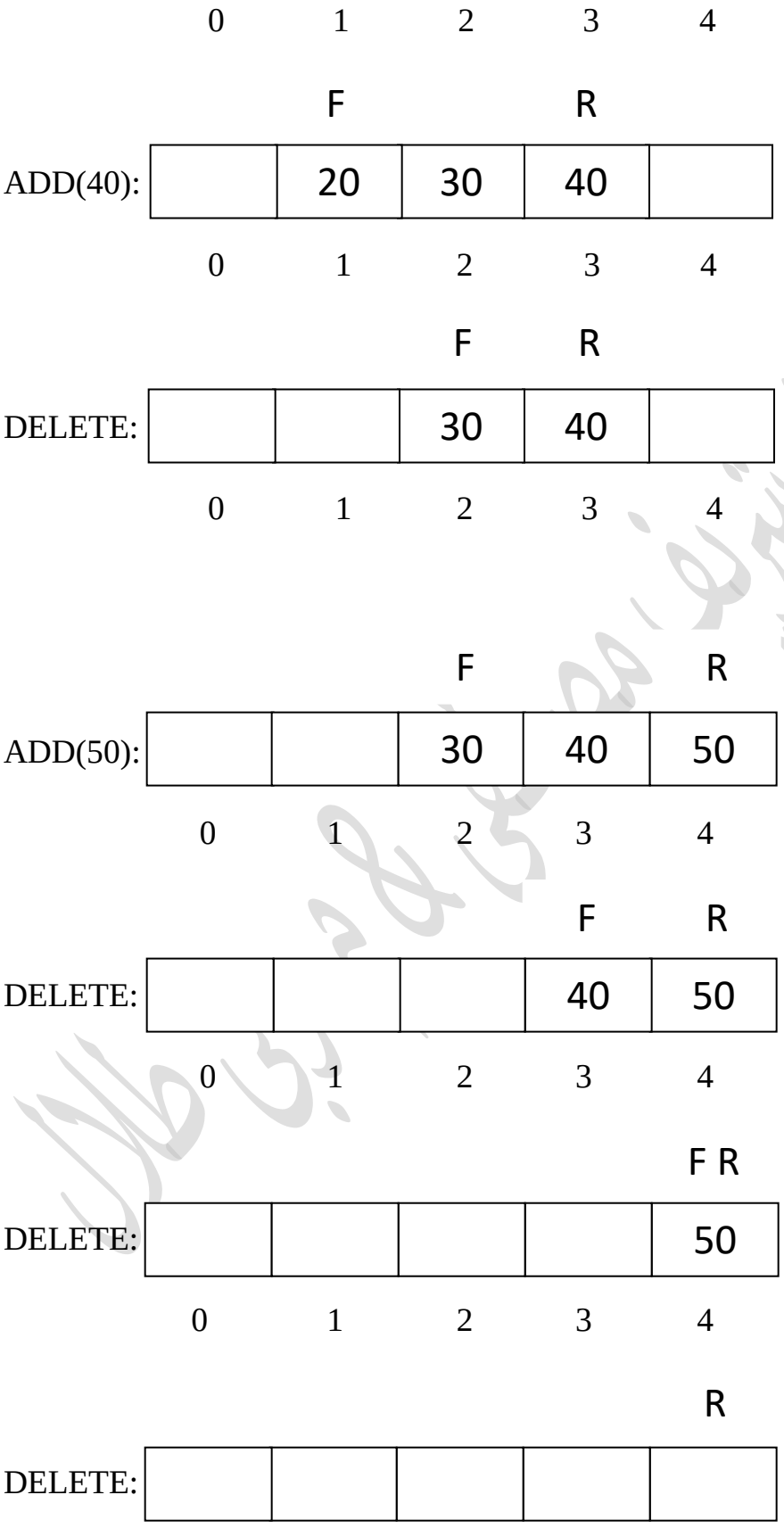


DELETE:



ADD(30):




0 1 2 3 4
F R
ADD(40):
20 30 40
0 1 2 3 4
F R
DELETE:
30 40
0 1 2 3 4
F R
ADD(50):
30 40 50
0 1 2 3 4
F R
DELETE:
40 50
0 1 2 3 4
F R
DELETE:
50
0 1 2 3 4
R
DELETE:

0 1 2 3 4

Tutorial:

Suppose having a queue with five locations. Draw figures to represent the contents of the queue and the location of front and rear pointers after performing the following add and delete operations:

Queue Initialization, ADD(A), DELETE, ADD(B), ADD(C), ADD(D), DELETE, DELETE, ADD(E), DELETE.