CHAPTER 5

DATA STRUCTURES AND ALGORITHMS

Queue Data Structure

CONTENTS

- Queue definition,
- Operations on queue,
- Queue implementation
  - Using array,
  - Using linked lists,
- Circular queue,
- Priority queue
- Applications,

5.1. QUEUE

- A queue is a list in which insertions ***are permitted only at one end of the list*** called its **rear/tail**, and ***all deletions are constrained to the other end*** called the **front/head** of the queue.
- 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 (FIFO)** methodology
- Real-world examples,
  - a single-lane one-way road

CONT..

- Queues are one of the most common data processing structures.
- They are frequently used in most system software such as ***operating systems***, ***network*** and ***database implementations***, and other areas.
- As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures.

5.2. OPERATIONS ON QUEUE

Basic Operations

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

Supportive operations

- **getFront()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.

ENQUEUE OPERATION

- Queues maintain two data pointers, front and rear.
- 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.

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, increment front pointer and access the data where front is pointing.
  - Step 4 – remove the data.
  - Step 5 – Return success.

5.3. QUEUES USING ARRAY

- The array-based queue is somewhat tricky to implement effectively.
- ***Create:*** *This operation should create an empty queue.*

```
#define max 50
int Queue[max];
int Front = Rear = -1;
```

Here max is the maximum initial size that is defined.

- ***Is_Empty:*** *This operation checks whether the queue is empty or not.This is confirmed by comparing the values of* ***Front*** *and* ***Rear****.*

```
bool Is_Empty()
{
   if(Front == Rear)
      return 1;
   else
      return 0;
}
```

CONT.

- ***Is_Full:*** *When Rear points to the last location of the array, it indicates that the queue is full*

```
bool Is_Full()
{
   if(Rear == max - 1)
      return 1;
   else
      return 0;
}
```

- ***Add:*** *This operation adds an element in the queue if it is not full. As Rear points to the last element of the queue, the new element is added at the* ***(rear + 1)th location****.*

```
void Add(int Element)
{
   if(Is_Full())
      cout << "Error, Queue is full";
   else
       Queue[++Rear] = Element;
}
```

CONT.

- ***Delete:*** *This operation deletes an element from the front of the queue and sets Front to point to the next element. We should first increment the value of Front and then remove the element.*

```
int Delete()
{
   if(Is_Empty())
      cout << "Sorry, queue is Empty";
   else
      return(Queue[++Front]);
}
```

- **getFront**: returns the element at the front, but unlike delete, this does not update the value of Front.

```
int getFront()
{
   if(Is_Empty())
      cout << "Sorry, queue is Empty";
   else
      return(Queue[Front + 1]);
}
```

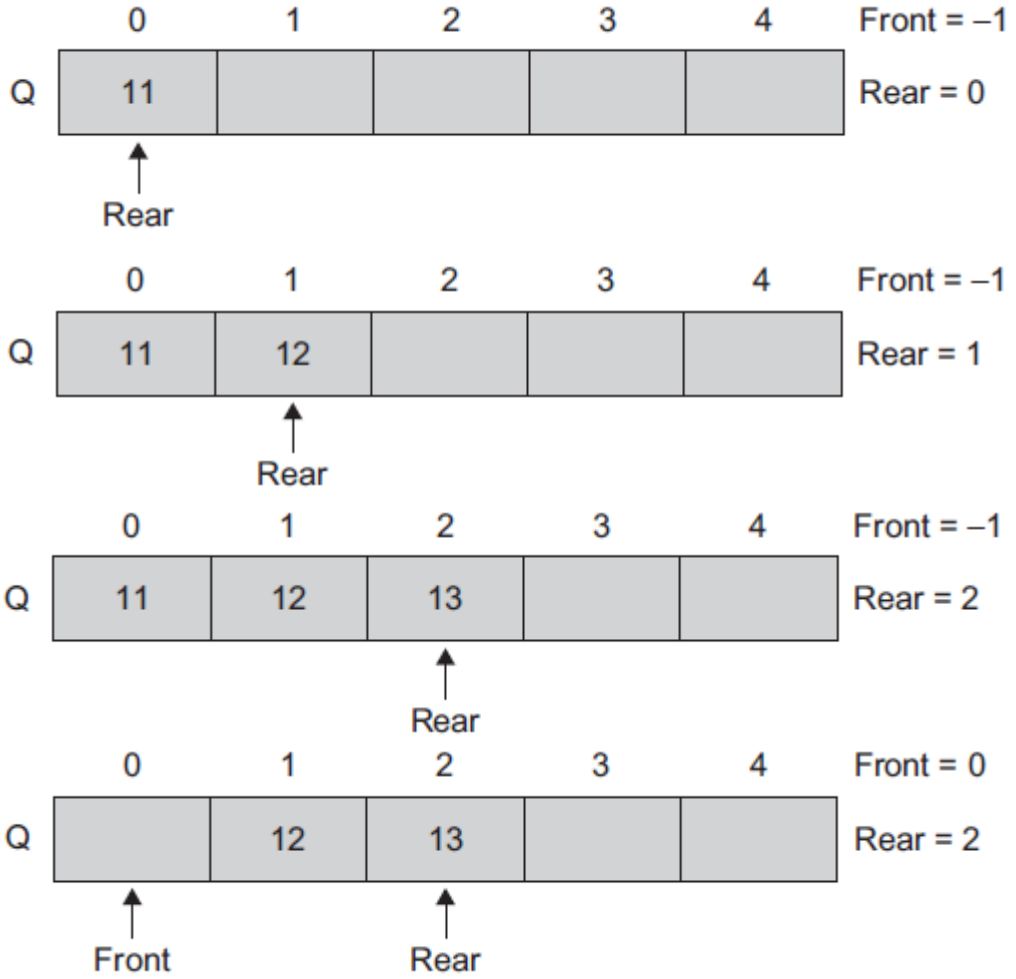
Let *Q* be an empty queue with `Front = Rear = -1. Let max = 5.`

Front = −1

Rear = −1

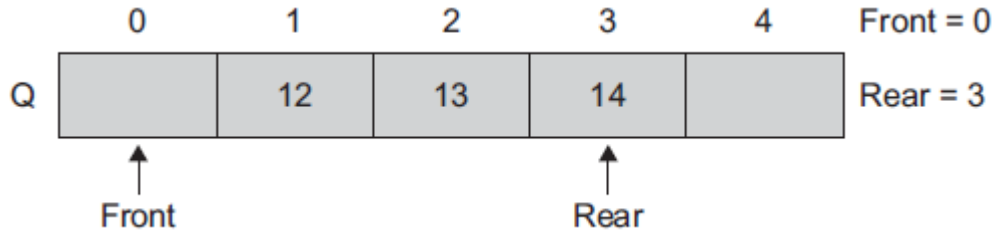
Consider the following statements:

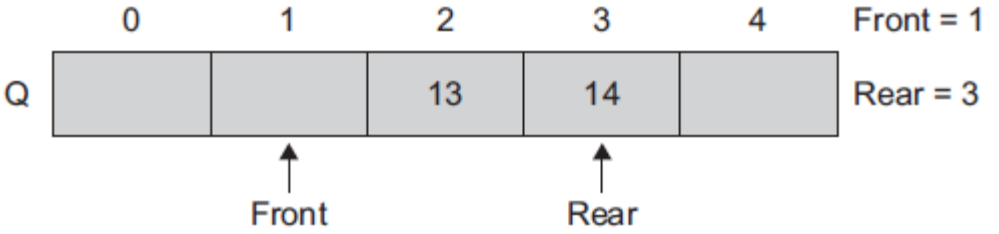
1. **Enqueue (11)**
2. **Enqueue (12)**
3. **Enqueue (13)**
4. **Dequeue ()**
5. Enqueue (14)
6. Dequeue ()
7. Dequeue ()
8. Dequeue ()
9. Dequeue ()
10. Enqueue (15)
11. Enqueue (16)

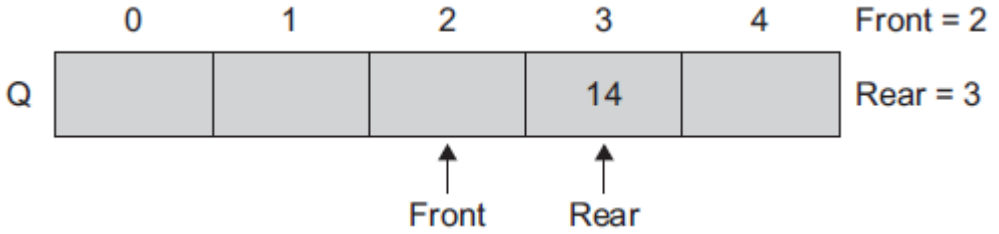


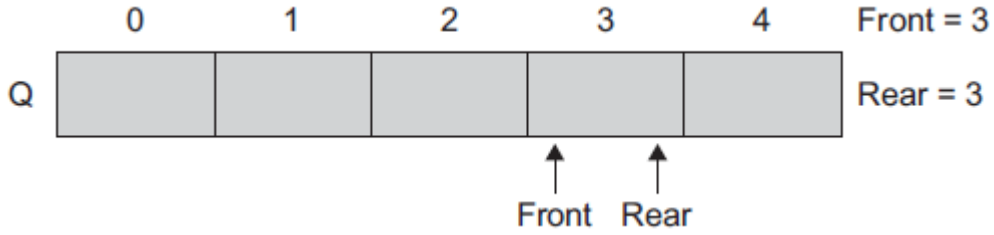
Consider the following statements:

1. Enqueue (11)
2. Enqueue (12)
3. Enqueue (13)
4. Dequeue ()
5. **Enqueue (14)**
6. **Dequeue ()**
7. **Dequeue ()**
8. **Dequeue ()**
9. **Dequeue ()**
10. **Enqueue (15)**
11. **Enqueue (16)**

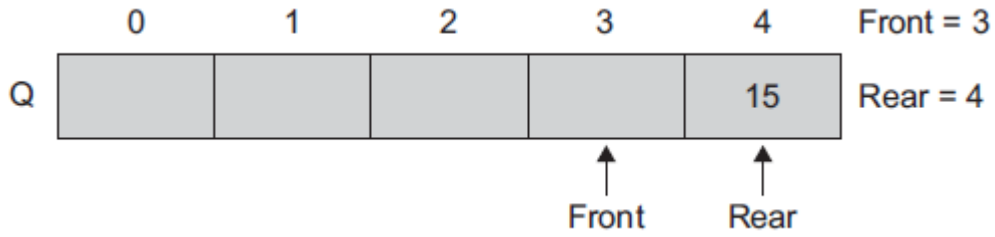








Here we get the `Queue_empty` error condition as `Front = Rear = 3`



This means that the implementation needs to be modified.

This statement will generate the message `Queue_Full` because `Rear = 4`.

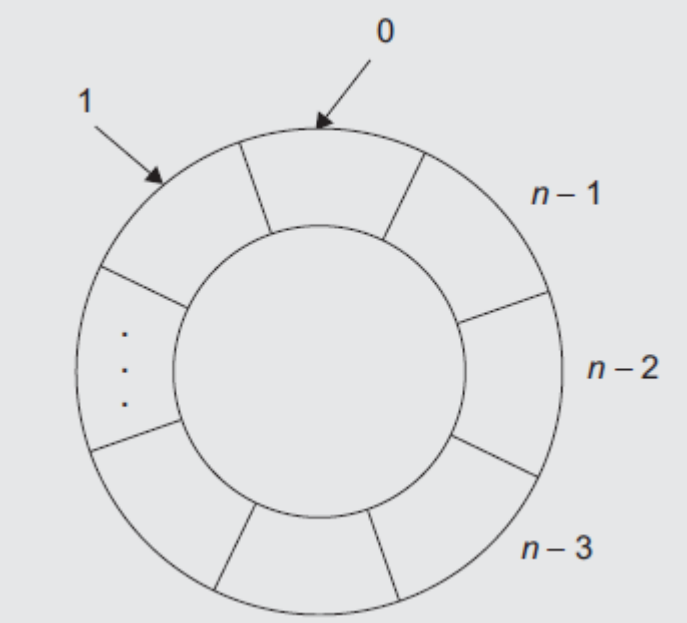
DRAWBACKS OF LINEAR QUEUE

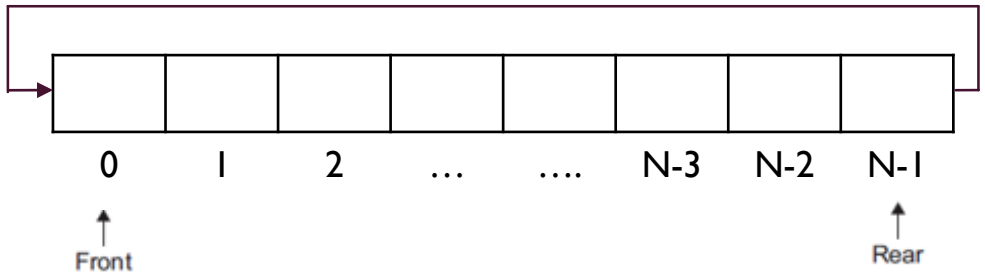
1. The linear queue is of a **fixed size**.
2. An arbitrarily declared maximum size of queues leads to **poor utilization of memory**.
3. Array implementation of linear queues leads to the **Queue_Full state** even though the queue is not actually full.
4. To avoid this, we need to **move the entire queue to the original start location** (if there are empty locations) so that the first element is at the 0^{th} location and Front is set to -1 (rear= rear-1).

This is obviously not a feasible solution as it is time consuming and involves a lot of data movement. This becomes impractical, especially when the queue is of a large size.

CIRCULAR QUEUE

- Allows the queue to **wraparound** upon reaching the end of the array eliminates drawbacks of linear queue.
- As we go on adding elements to the queue and reach the end of the array, the next element is stored in the first slot of the array if it is empty.
- The queue is said to be full only when there are n elements in the queue.

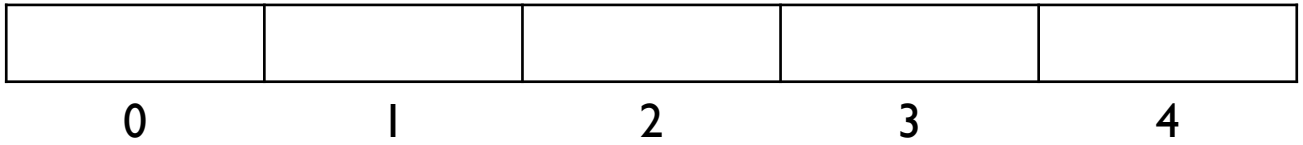




Enqueue index = (rear + 1) % Max

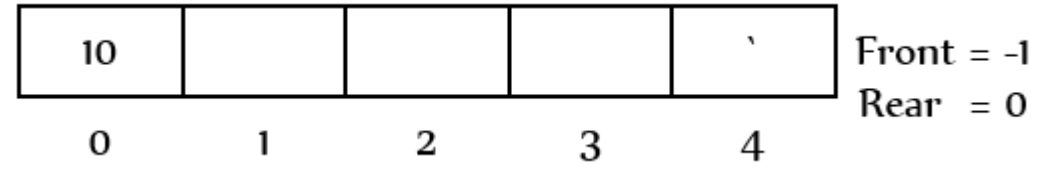
Dequeue index = (front + 1) % Max

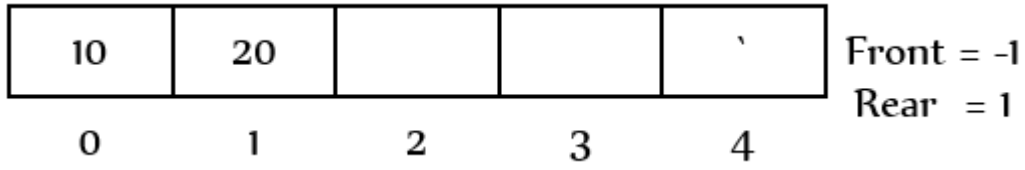
Let *Q* be an empty queue with `Front = Rear = -1. Let max = 5.`

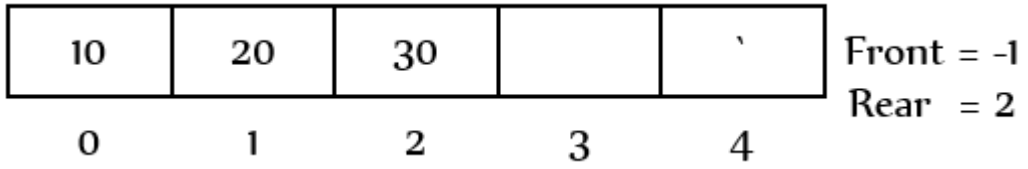


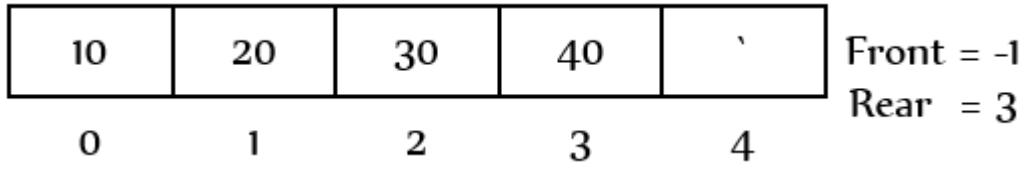
Consider the following statements:

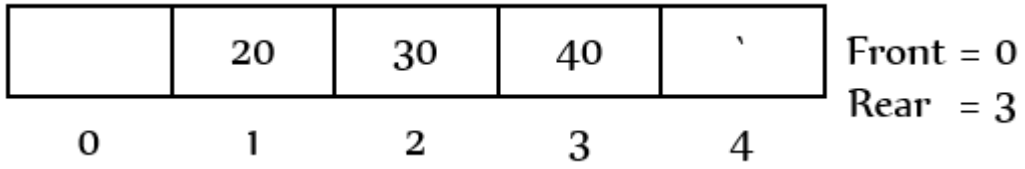
1. **Enqueue (10)**
2. **Enqueue (20)**
3. **Enqueue (30)**
4. **Enqueue (40)**
5. **Dequeue ()**
6. Dequeue ()
7. Enqueue (50)
8. Enqueue (60)
9. Enqueue (70)
10. Enqueue (80)



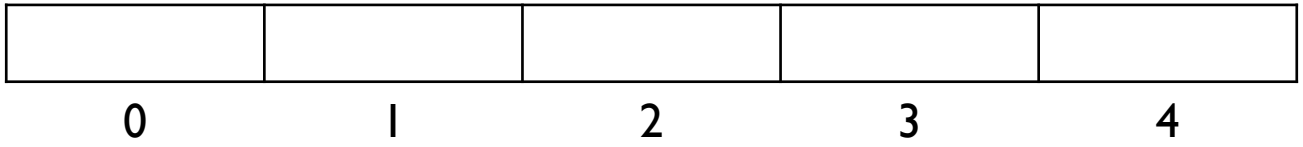






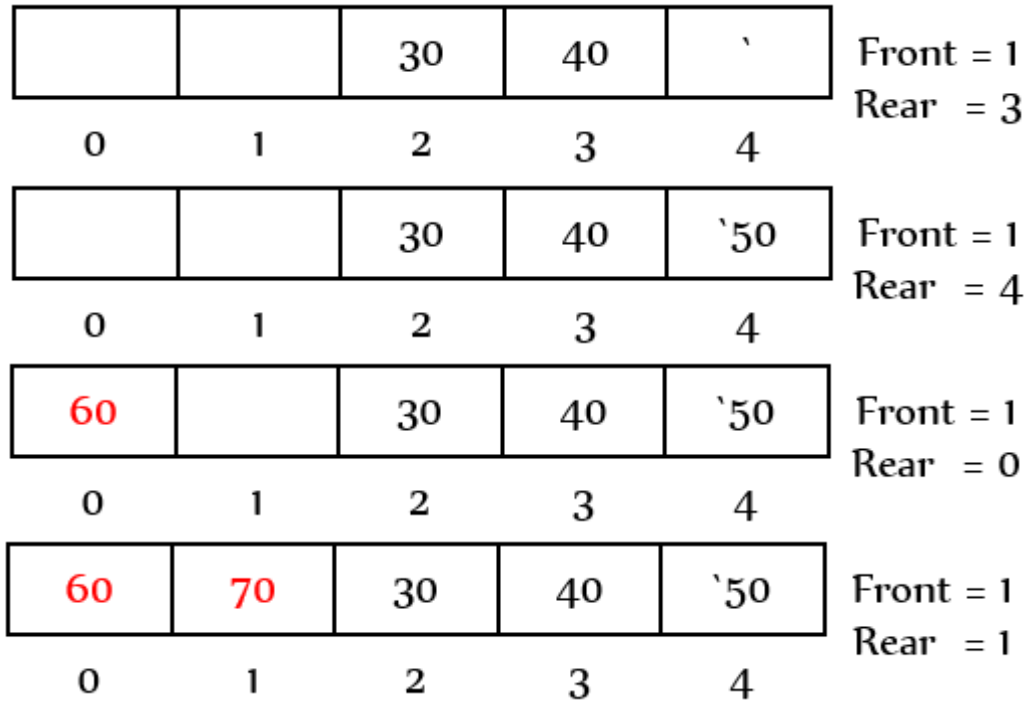


Let *Q* be an empty queue with `Front = Rear = -1. Let max = 5.`



Consider the following statements:

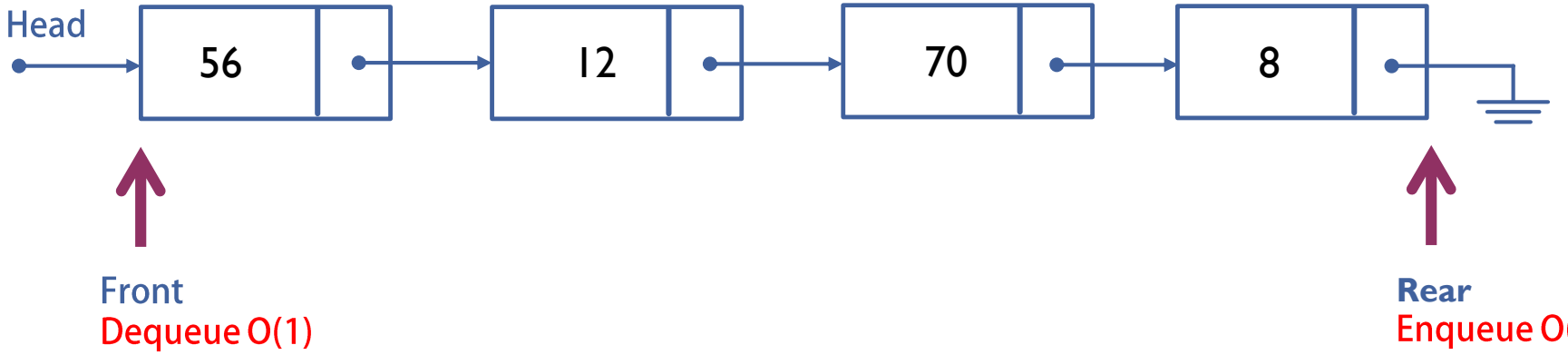
1. Enqueue (10)
2. Enqueue (20)
3. Enqueue (30)
4. Enqueue (40)
5. Dequeue ()
6. **Dequeue ()**
7. **Enqueue (50)**
8. **Enqueue (60)**
9. **Enqueue (70)**
10. **Enqueue (80)**



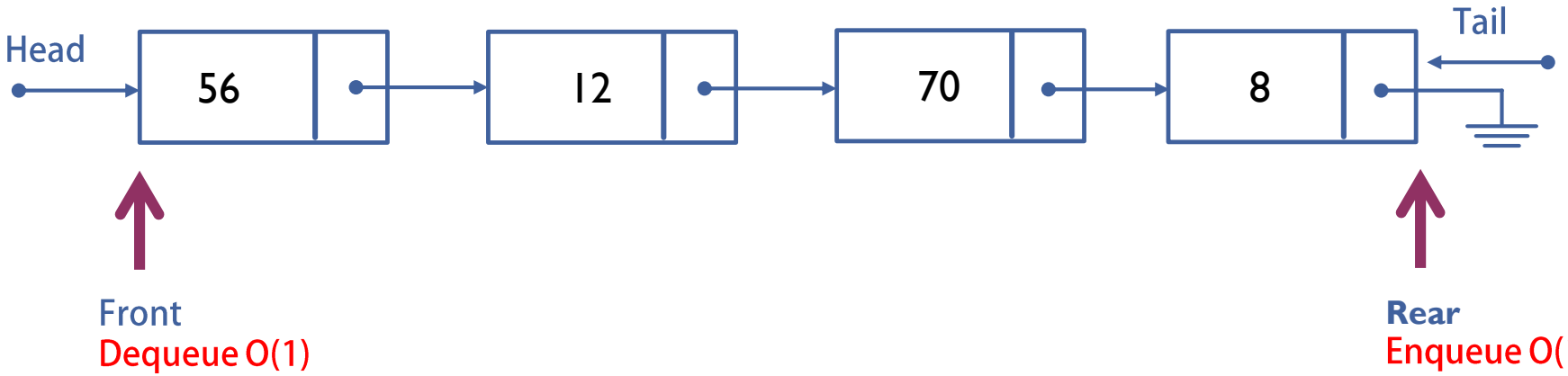
`Enqueue (80) will generate the message Queue_Full because There are 5 elements in the queue.`

QUEUE IMPLEMENTATION USING LINKED LIST

- We can use either side of the linked list to enqueue while the other side is used to dequeue.



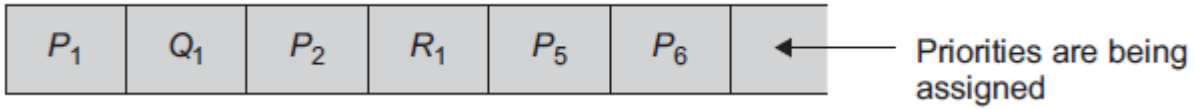
To execute both operations in constant time we have to maintain tail pointer in addition to head pointer.



PRIORITY QUEUE

- A priority queue is a collection of a finite number of prioritized elements.
- Elements can be inserted in any order in a priority queue, but when an element is removed from the priority queue, it is always the one with the highest priority.
- The following rules are applied to maintain a priority queue:
  1. The element with a higher priority is processed before any element of lower priority.
  2. If there were elements with the same priority, then the element added first in the queue would get processed first.

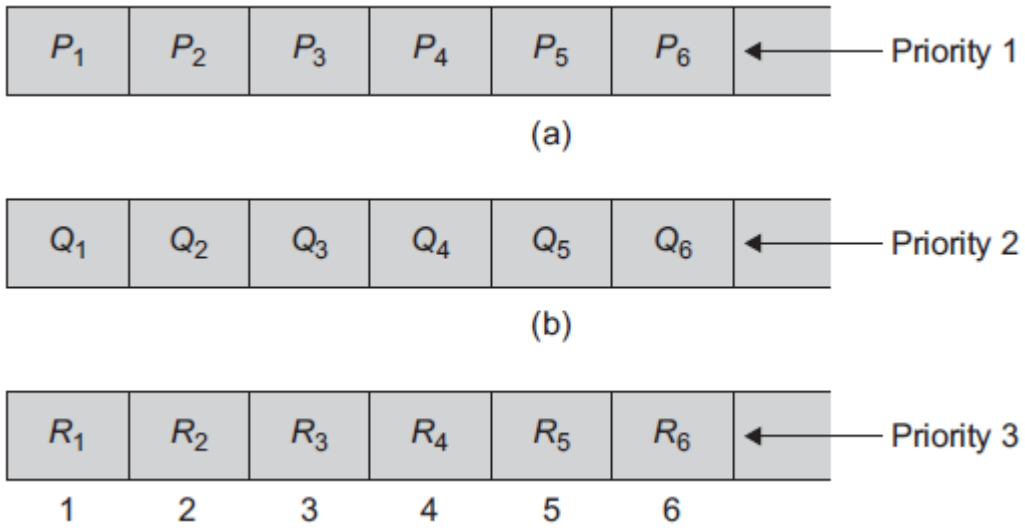
- Priority queues are used for implementing job scheduling by the operating system where jobs with higher priority are to be processed first.
- A list of jobs carried out by a multitasking operating system; each background job is given a priority level.
- Suppose in a computer system, jobs are assigned three priorities, namely, P, Q, R as first, second, and third, respectively.



PRIORITY QUEUES IMPLEMENTATION

Implementation method 1

- Implemented using three separate queues, each following the FIFO behavior strictly as shown in bellow.



In this example, jobs are always removed from the front of the queue.

The elements in the second queue are removed only when the first queue is empty, and

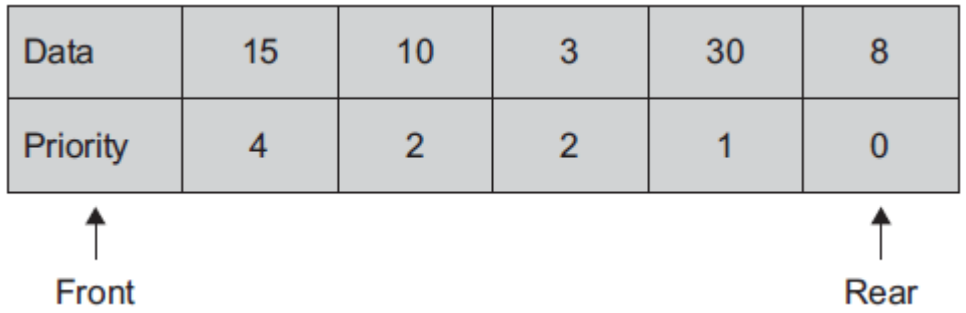
The elements from the third queue are removed only when the second queue is empty, and so on.

PRIORITY QUEUES IMPLEMENTATION

Implementation method 2

- The second way of priority queue implementation is by using a structure for a queue.

```
Struct QueueElement
{
    int Data;
    int priority;
};
```



- The highest priority element is at the front and that of the lowest priority is at the rear.

CONT..

- The two ways to implement a priority queue are sorted list and unsorted list.

Sorted list: A sorted list is characterized by the following features:

- Advantage—Deletion is easy; elements are stored by priority, so just delete from the beginning of the list.
- Disadvantage—Insertion is hard; it is necessary to find the proper location for insertion.
- A linked list is convenient for this implementation such as the list in Fig. 5.9.

Unsorted list: An unsorted list is characterized by the following features:

- Advantage—Insertion is easy; just add elements at the end of the list.
- Disadvantage—Deletion is hard; it is necessary to find the highest priority element first.
- An array is convenient for this implementation.

APPLICATIONS OF QUEUES

- Queues are also very useful in a time-sharing computer system where many users share a system simultaneously.
- Whenever a user requests the system to run a particular program, the operating system adds the request at the end of the queue of jobs waiting to be executed.
- Now, when the CPU is free, it executes the job that is at the front of the job queue.
- Similarly, there are queues for shared I/O devices too. Each device maintains its own queue of requests.
- Queues are also used in simulations and different problem solving operations. Read the reference book for more information on applications of queues.