Queue

Queue ADT

- A **Queue** is a data structure that has two ends:
  - Elements are added at one end called "**rear**".
  - And removed from the other end called "**front**".

- Insertions and deletions follow **First-in First-out** (FIFO) scheme (principle).
  - It means that the element added last will be removed first.

Queue ADT

- **Main operations**
  - **enqueue(object): I**nsert element at **rear.**
  - **object dequeue():** Remove and returns element at **front**.
- **Auxiliary operations**
  - object front(): returns front element without removing it.
  - integer size(): returns number of elements stored.
  - boolean isEmpty(): returns whether no elements are stored.

Applications of Queues

- Direct
  - Waiting lines.
  - Access to shared resources.
  - Hold jobs for a printer.
  - The most common application is in client-server models
    - Multiple clients may be requesting services from one or more servers
    - Some clients may have to wait while the servers are busy
    - Those clients are placed in a queue and serviced in the order of arrival
- Indirect
  - Auxiliary data structure for algorithms
  - Component of other data structures

Array-based Queue

- Add elements in an array **Q** of capacity(size) ***N.***
- Two Variables:
  - *front* that points to the beginning of the Queue.
  - *rear* that points to the end of the Queue.

Enqueue and Dequeue Algorithms

```
Algorithm Enqueue(Element):
    if isFull then
        throw Full Queue Exception
    else
        Q[rear] ← element
        rear ← rear + 1
```

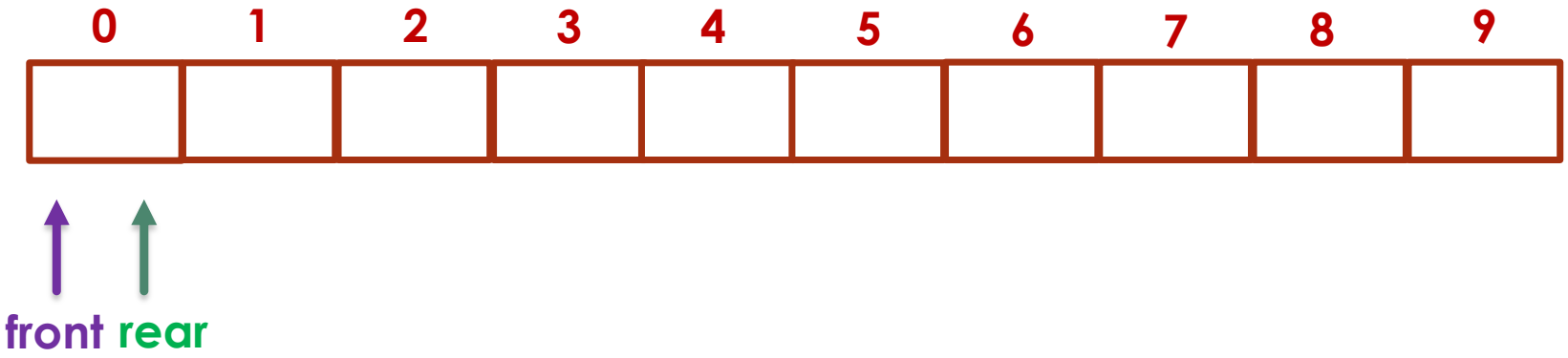
Run time: O(1)

```
Algorithm Dequeue():
    if isEmpty then
        throw Empty Queue Exception
    else
        item ← Q[front]
        Q[front] ← Null
        front ← front+1
        return item
```

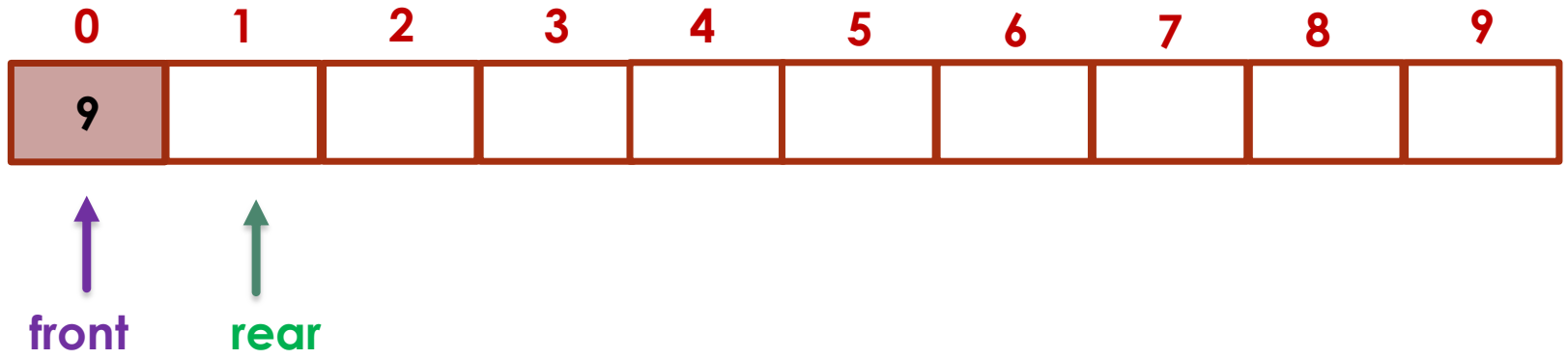
Run Time: O(1)

Queue Operations - Example

- Queue Q, N=10

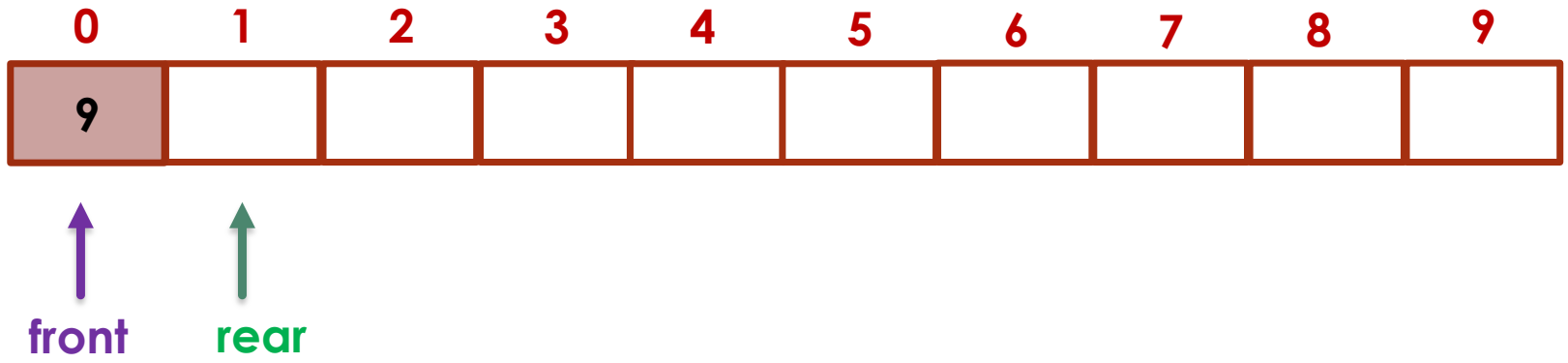



```
Enqueue (9)
{
    if isFull Then
        "Queue is Full"
    else
        Q[rear]=9
        rear=rear+1
}
```

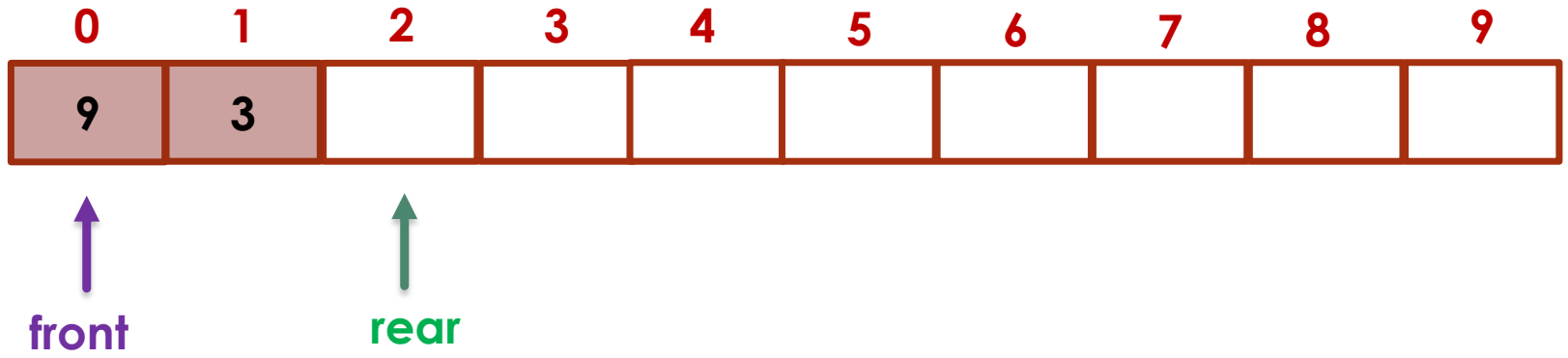


Queue Operations - Example

- Queue Q, N=10

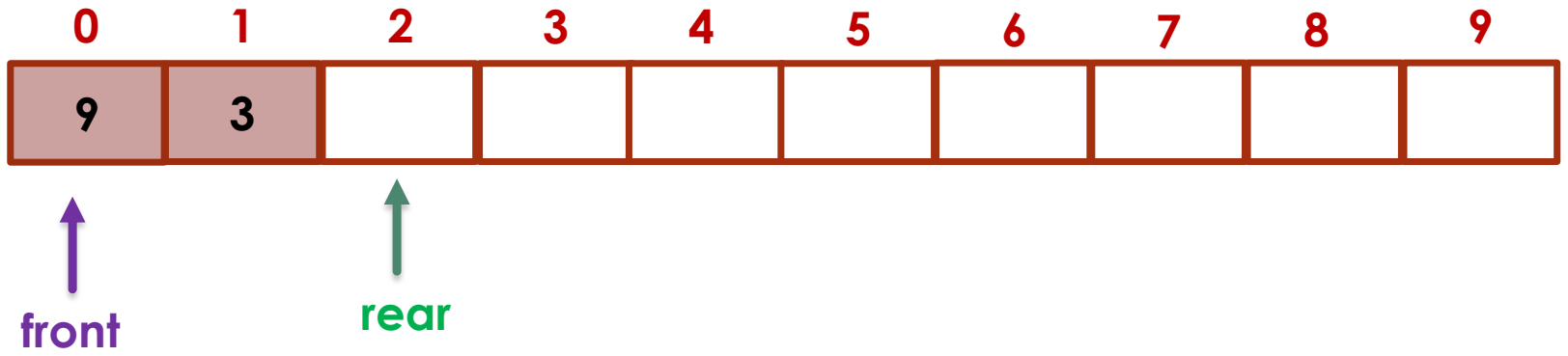



```
Enqueue (3)
{
    if isFull Then
        "Queue is Full"
    else
        Q[rear]=3
        rear=rear+1
}
```

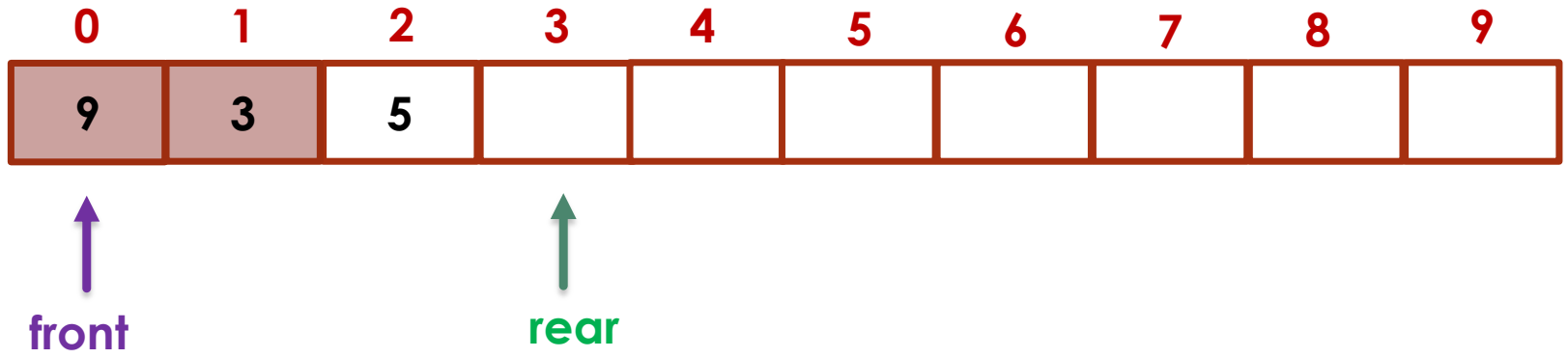


Queue Operations - Example

- Queue Q, N=10



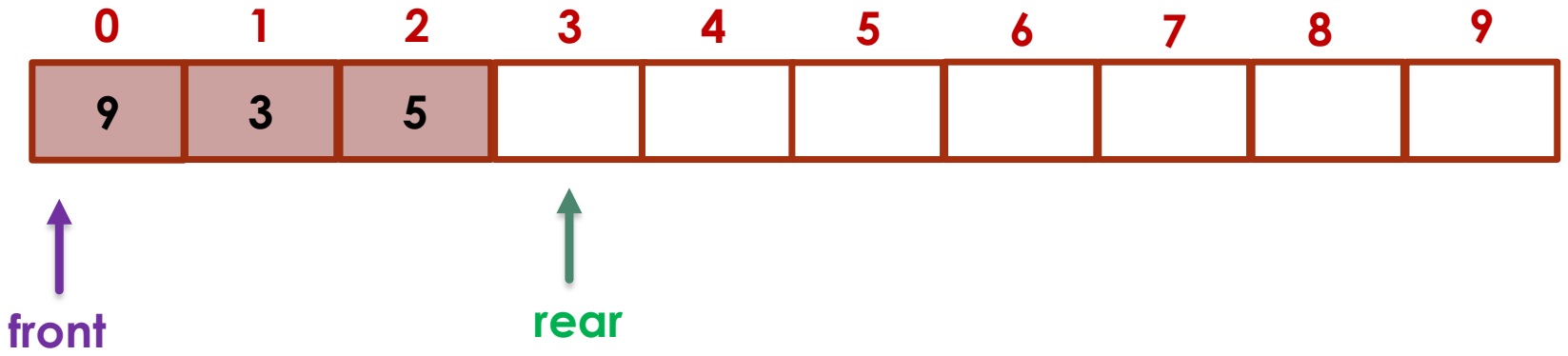

```
Enqueue (5)
{
    if isFull Then
        "Queue is Full"
    else
        Q[rear]=5
        rear=rear+1
}
```

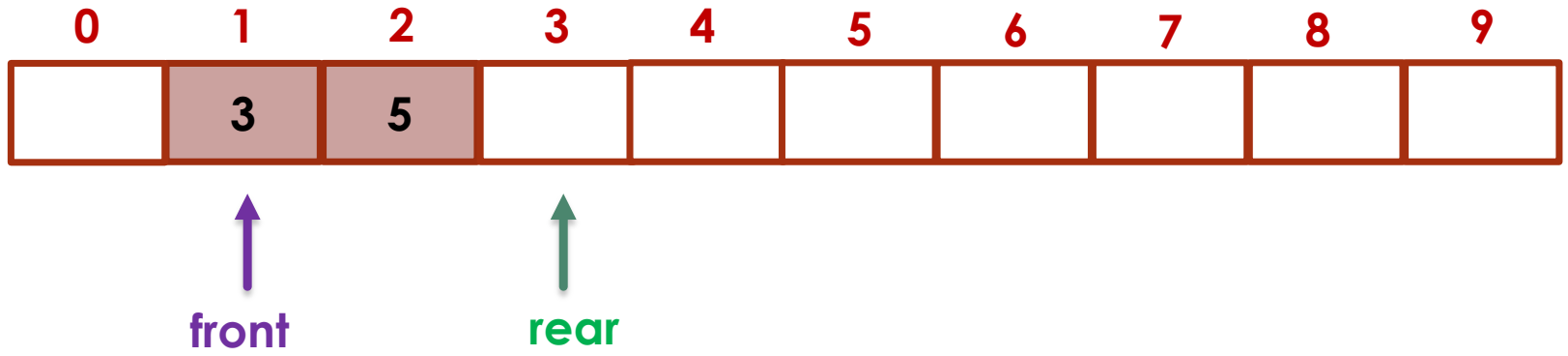


Queue Operations - Example

- Queue Q, N=10

```
Dequeue ()
{
    if isFull Then
        "Queue is empty"
    else
        item=Q[front]
        Q[front]=null
        front=front+1
        return item
}
```

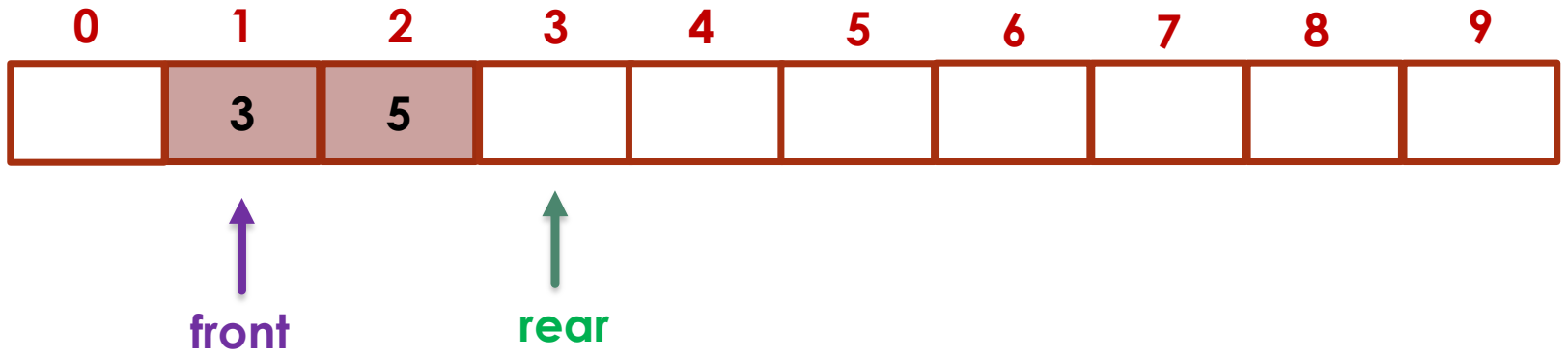


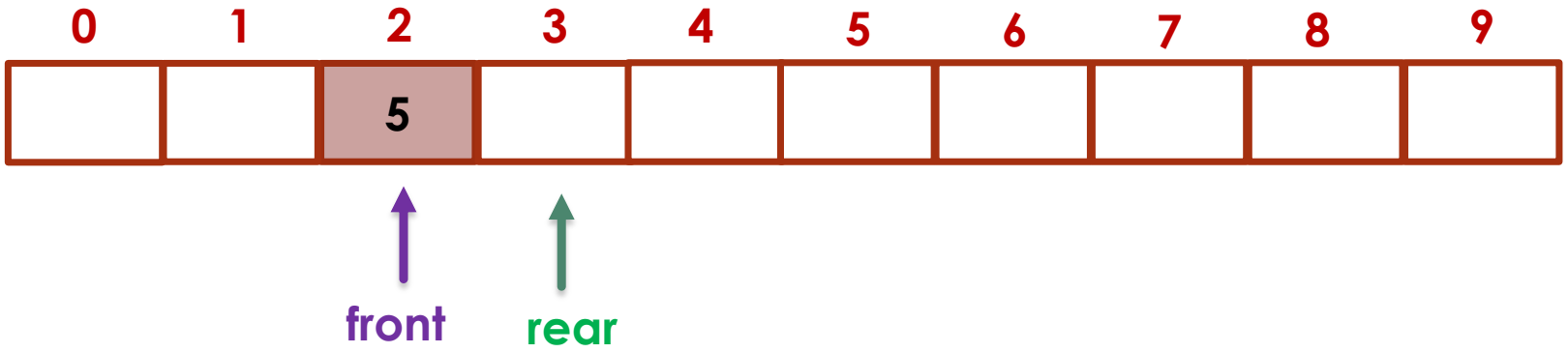


Queue Operations - Example

- Queue Q, N=10

```
Dequeue ()
{
    if isEmpty Then
        "Queue is empty"
    else
        item=Q[front]
        Q[front]=null
        front=front+1
        return item
}
```

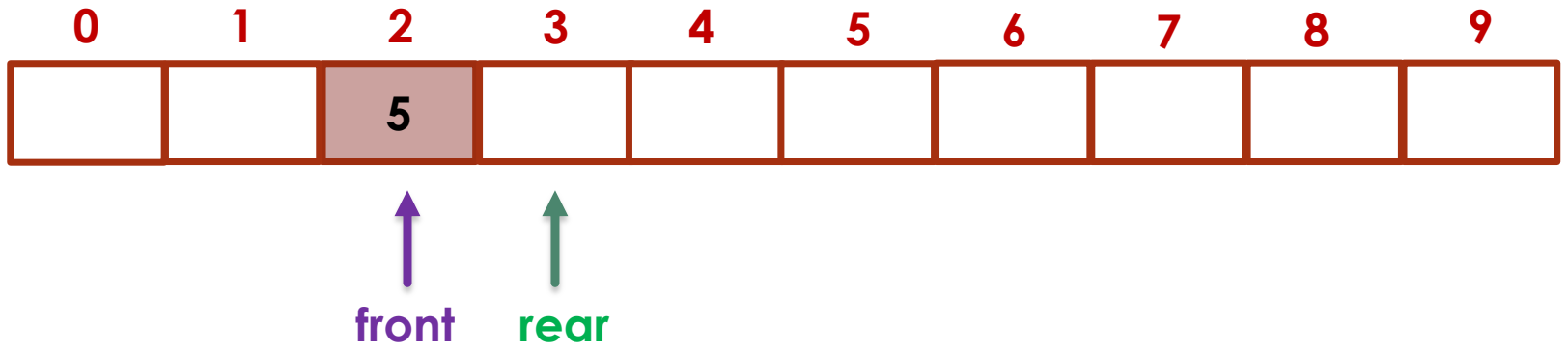


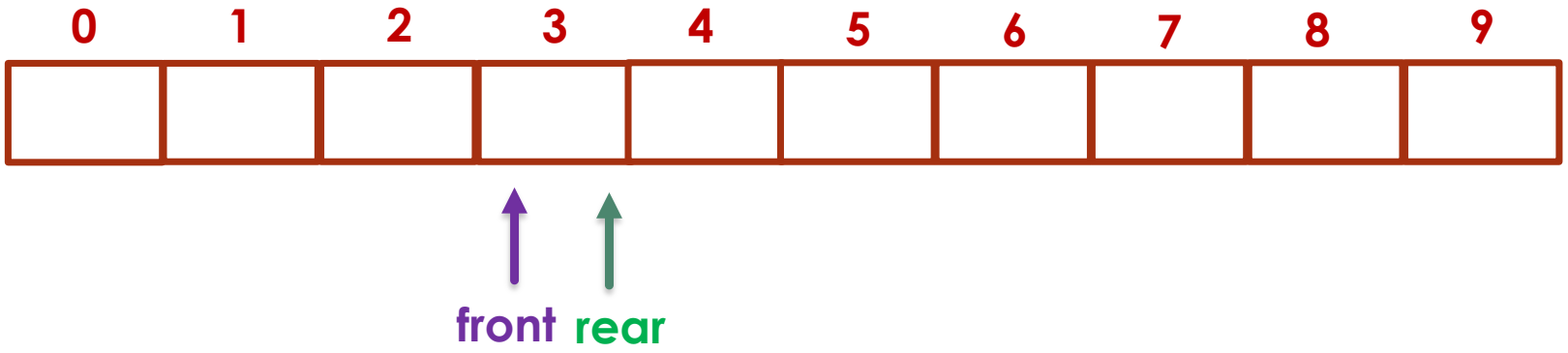


Queue Operations - Example

- Queue Q, N=10

```
Dequeue ()
{
    if isEmpty Then
        "Queue is empty"
    else
        item=Q[front]
        Q[front]=null
        front=front+1
        return item
}
```

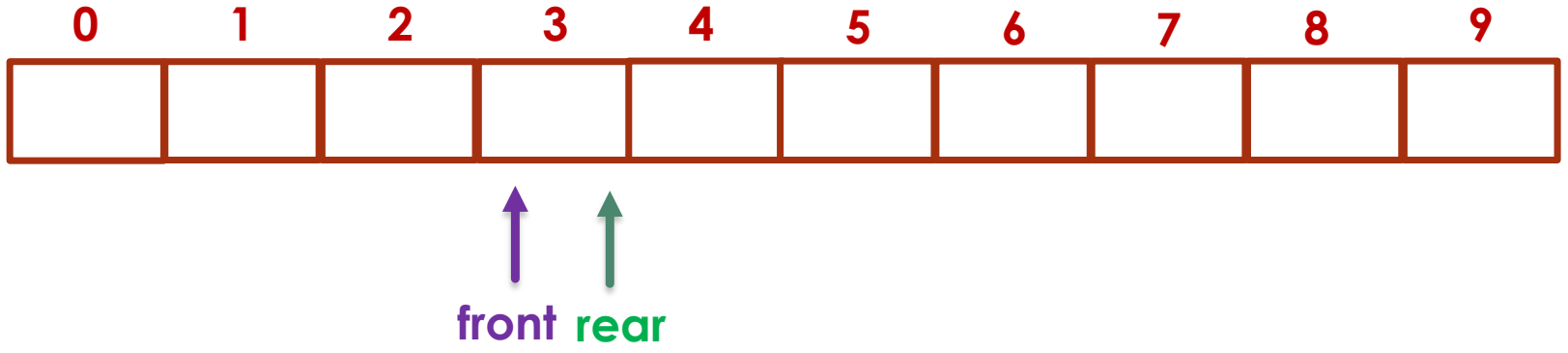


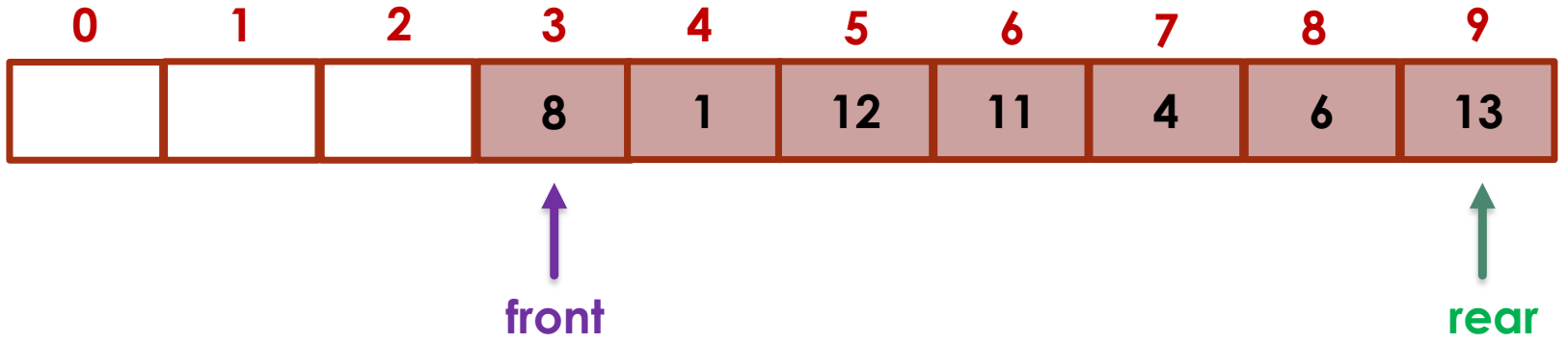


Queue Operations - Example

- **Queue Q, N=10**

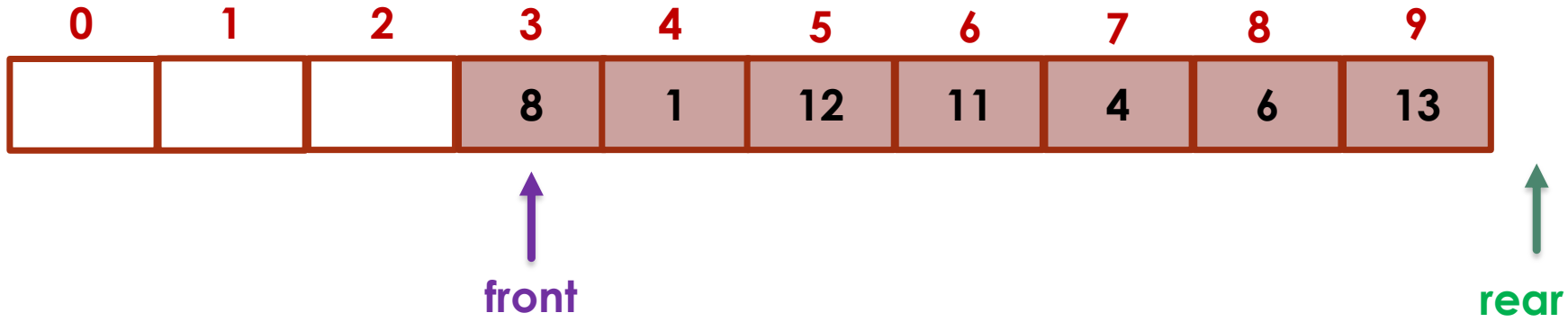
Enqueue (8)
Enqueue (1)
Enqueue (12)
Enqueue (11)
Enqueue (4)
Enqueue (6)
Enqueue (13)





Queue Operations - Example

- Queue Q, N=10



Now, let's add one more element 7.

```
Enqueue (7)
{
    if isFull Then
        "Queue is Full"
    else
        Q[rear]=7
        rear=rear+1
}
```

What is the problem?

Once queue becomes full, we can not insert the next element even if there is a space in front of queue. waste of memory in a Queue.

Circular Queue

To Solve the waste memory in Queue.

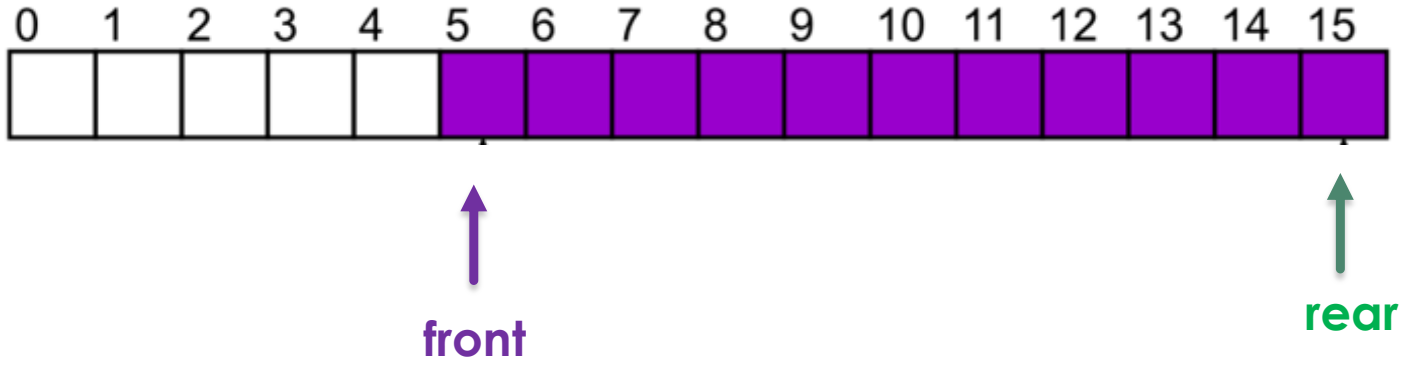
Circular Queue

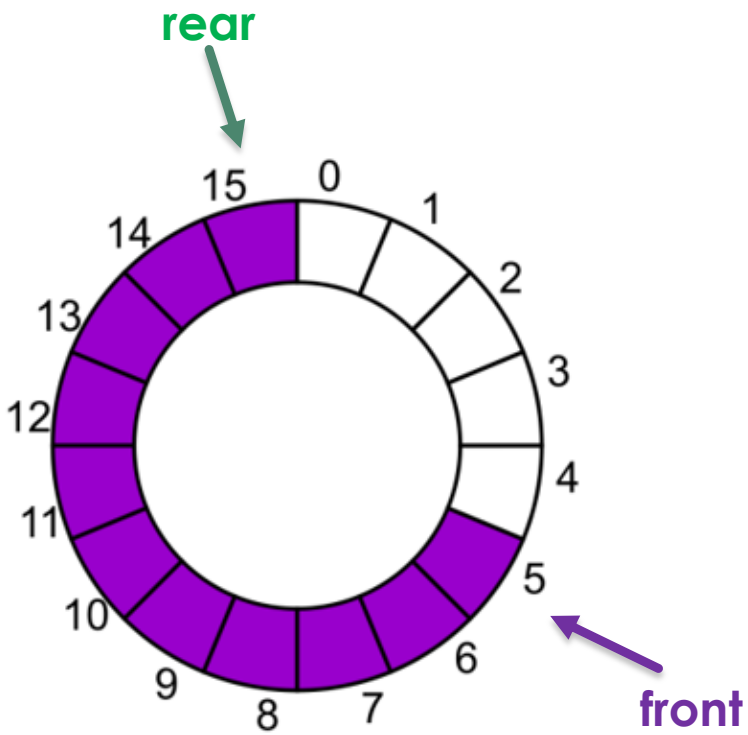
- Instead of viewing the array on the range 0, …, 15, consider the indices being cyclic:

…, 15, 0, 1, …, 15, 0, 1, …, 15, 0, 1, …

This is referred to as a circular array.

- view Q as a "circular array" that goes from Q [0] to Q [N-1] and then immediately back to Q [0] again.





Enqueue and Dequeue Algorithms

```
Algorithm Enqueue(Element):
    if isFull then
        throw Full Queue Exception
    else
        Q[rear] ← element
        rear ← (rear + 1) mod N
```

Run time: O(1)

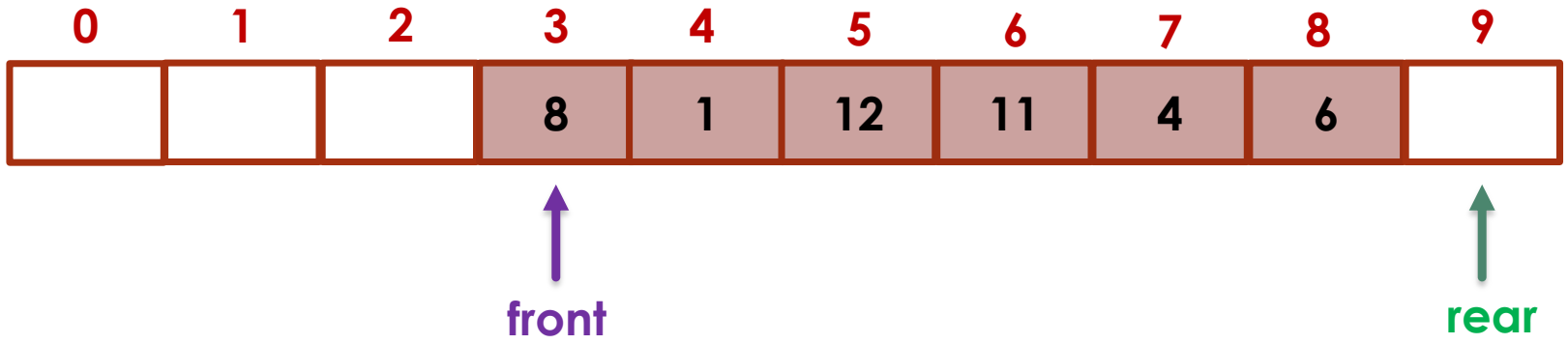
```
Algorithm Dequeue():
    if isEmpty then
        throw Empty Queue Exception
    else
        item ← Q[front]
        Q[front] ← Null
        front ← (front+1) mod N
        return item
```

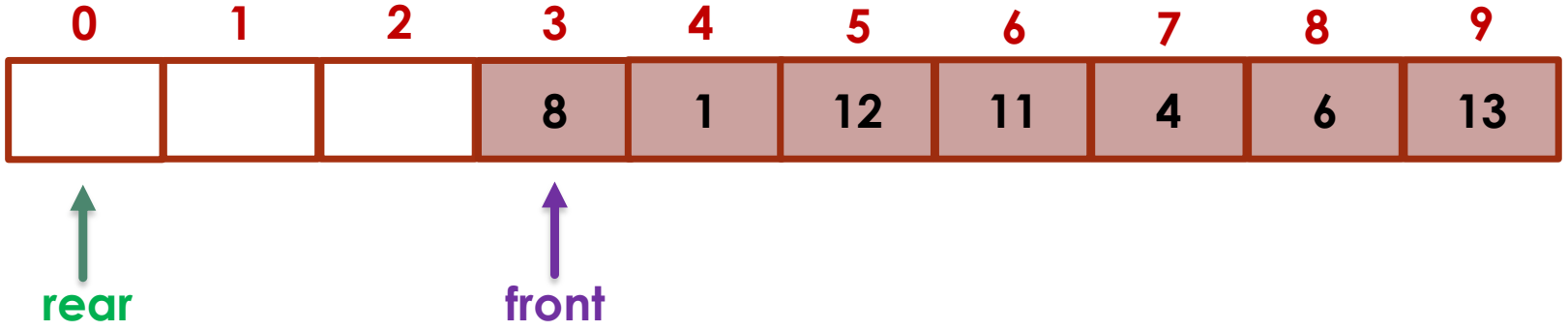
Run Time: O(1)

CQueue Operations - Example

- Queue Q, N=10

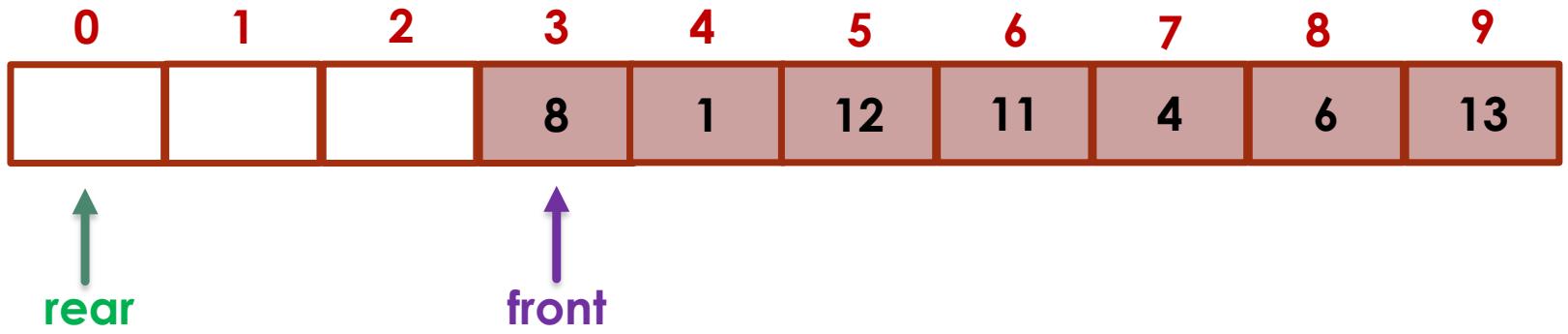
```
Enqueue (13)
{
    if isFull Then
        "Queue is Full"
    else
        Q[rear]=13
        rear=(rear+1) mod N
}
```



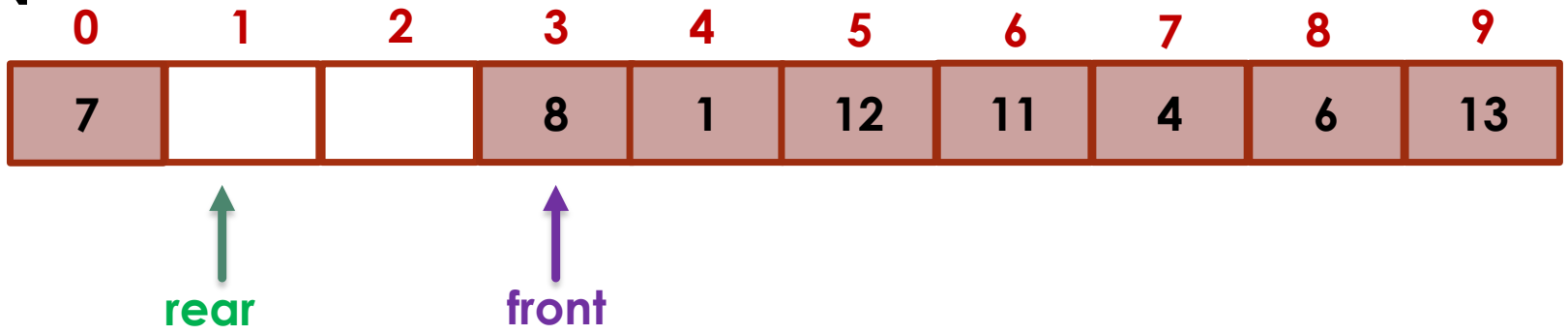


CQueue Operations - Example

- Queue Q, N=10



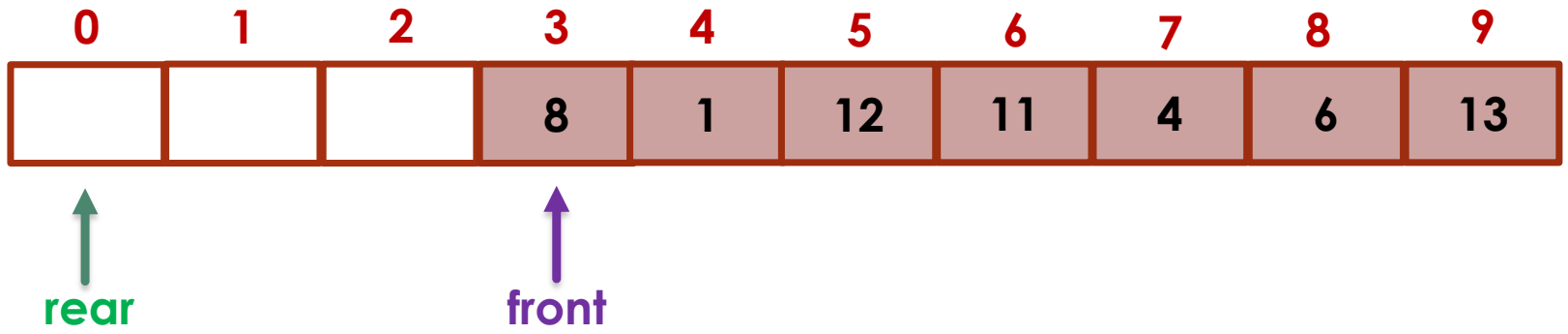

```
Enqueue (7)
{
    if isFull Then
        "Queue is Full"
    else
        Q[rear]=7
        rear=(rear+1) mod N
}
```

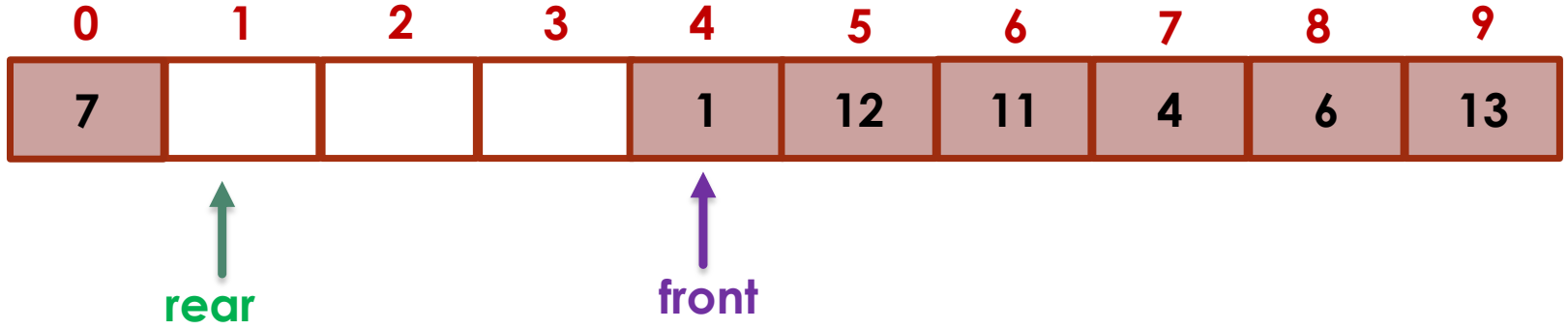


CQueue Operations - Example

- Queue Q, N=10

```
Dequeue ()
{
    if isEmpty Then
        "Queue is Empty"
    else
        item=Q[front]
        Q[front]=Null
        front=(front+1) mod N
}
```





Queue Implementation

- **Array**: We will use this first.
- **Linked Lists**: Later to be implemented with list.

Lab Assignment

- Implement the **Queue** in C++ using OOP.

Exercises

- Describe in pseudo-code an algorithm for reversing a queue Q. To access the queue, you are only allowed to use the methods of a queue ADT. **Hint**: Consider using an auxiliary data structure.

Exercises

- A linear list of elements in which deletion can be done from one end (front) and insertion can take place only at the other end (rear) is known as a?
  a) Queue.
  b) Stack.
  c) Tree.
  d) Linked list.
- A queue follows
  a) FIFO (First In First Out) principle.
  b) LIFO (Last In First Out) principle.
  c) Ordered array.
  d) Linear tree.