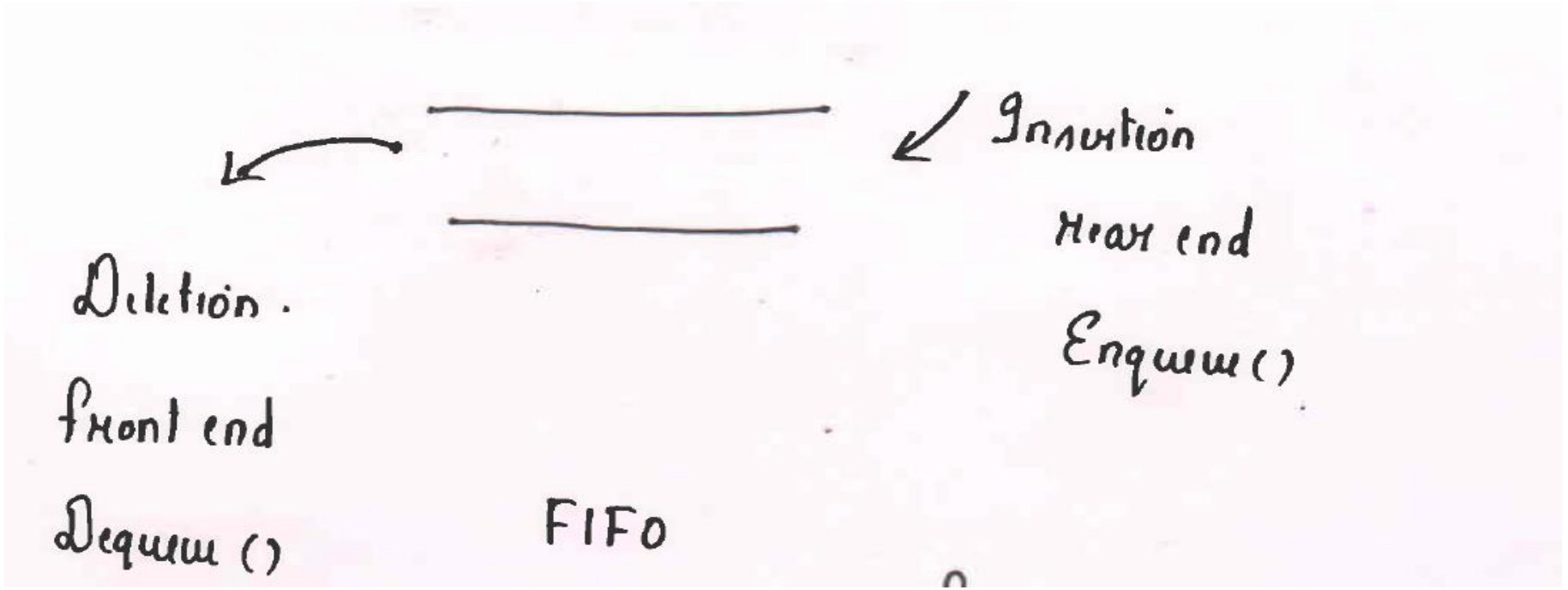
Queue ADT

- Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

Queue

Insertion and Deletion happen on different ends

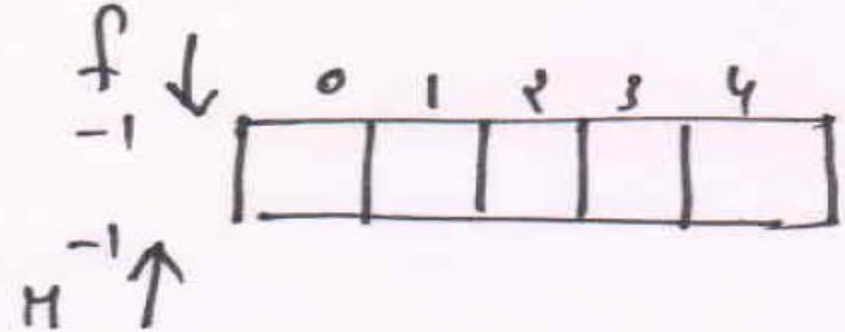


Queues maintain two data pointers, **front** and **rear**.

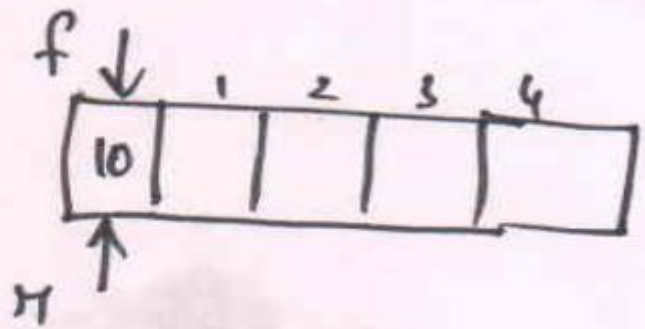
- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.
- **isempty()** − Checks if the queue is empty.
- **peek() or getFront()** − Gets the element at the front of the queue without removing it.

Array Based Implementation of a Queue

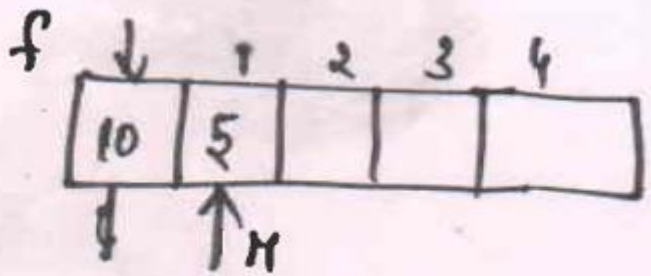
Array → Queue of size of 5.



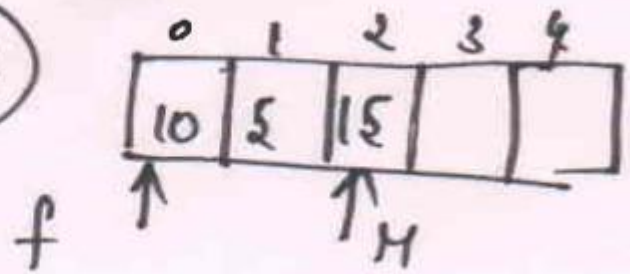
Enqueue (10)

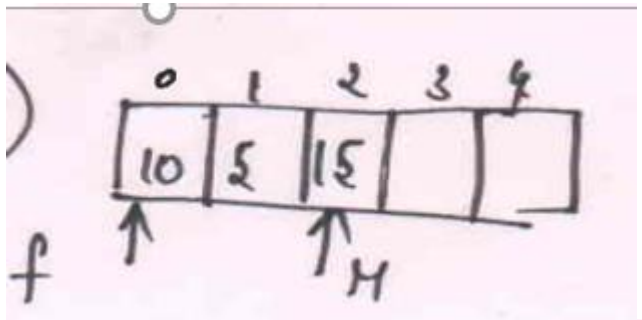


Enqueue (5)

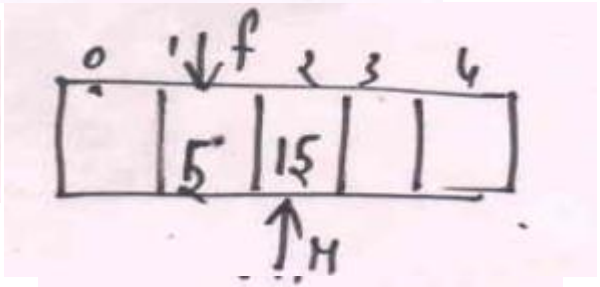


Enqueue (15)

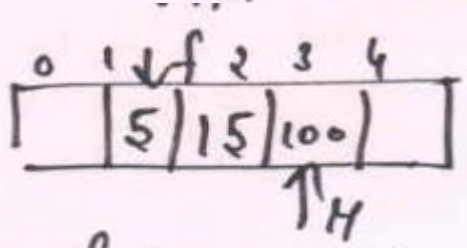




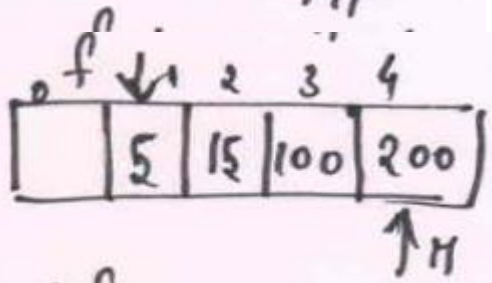
Dequeue()



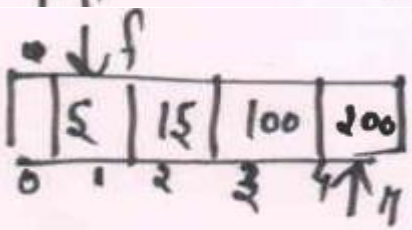
Enqueue(100)



Enqueue(200)

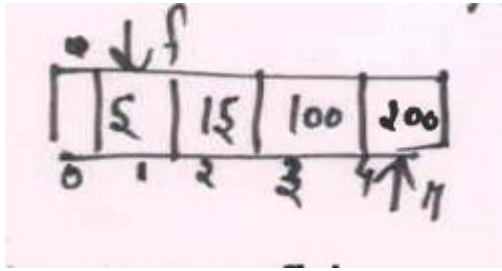


Enqueue(50)

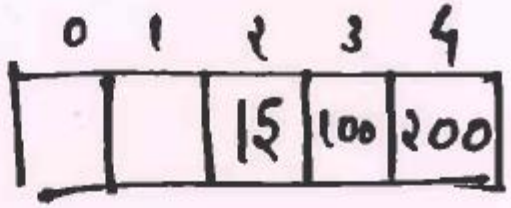


Rear = n-1 = 5-1 = 4

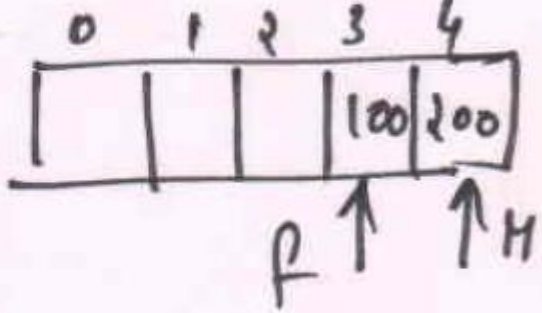
Queue full



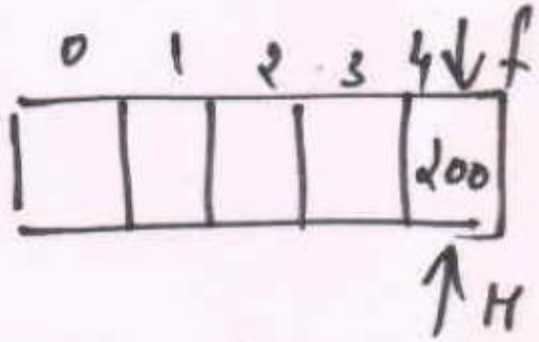
Deque()



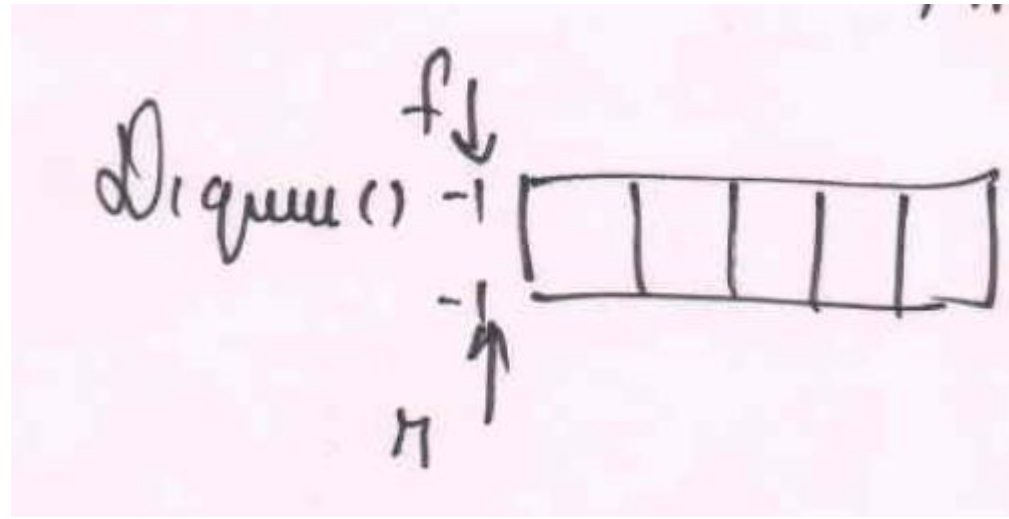
Deque()



Deque()



Deque()

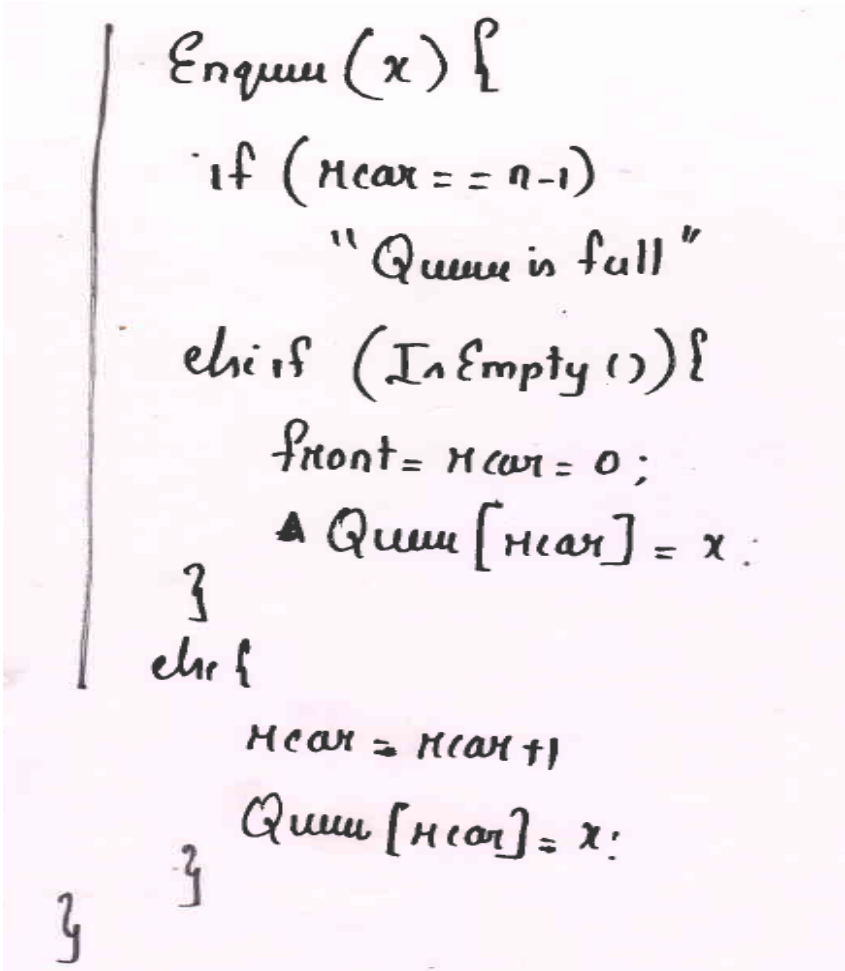


IsEmpty()

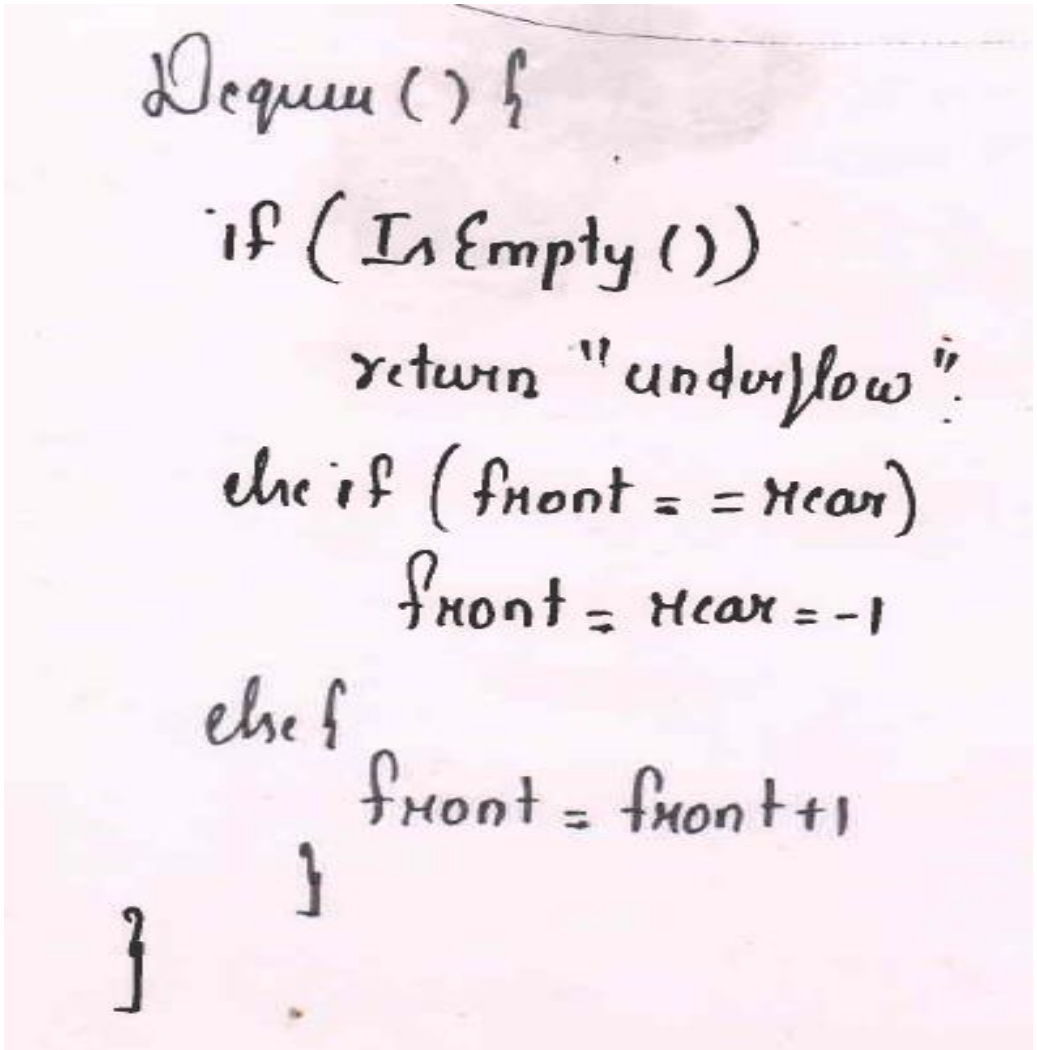
```
Boolean IsEmpty () {

    if (front == -1 && rear == -1)
        return true;
    else
        return false;
}
```

enqueue() – add (store) an item to the queue.

- Pointer affected is rear .

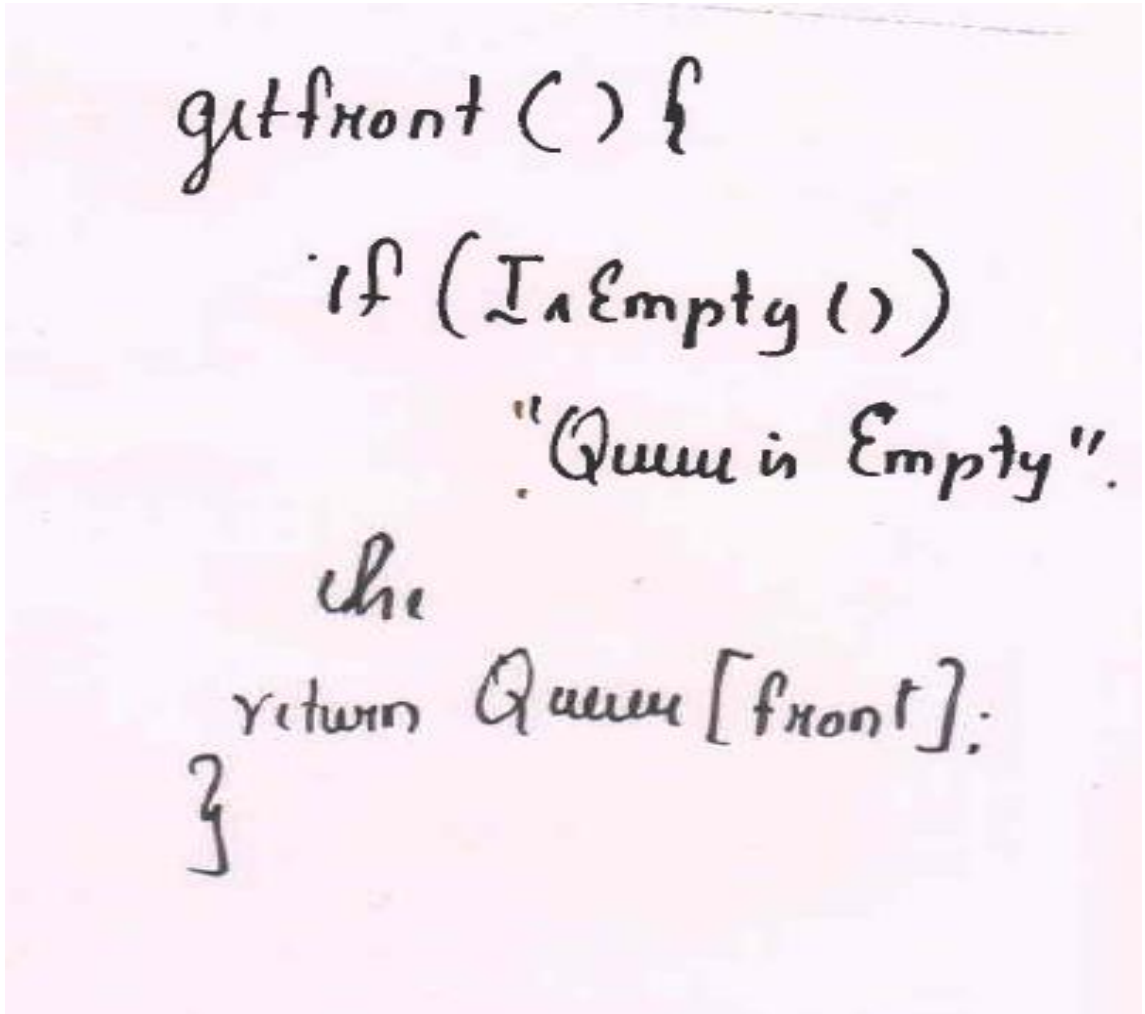


dequeue() – remove (access) an item from the queue.

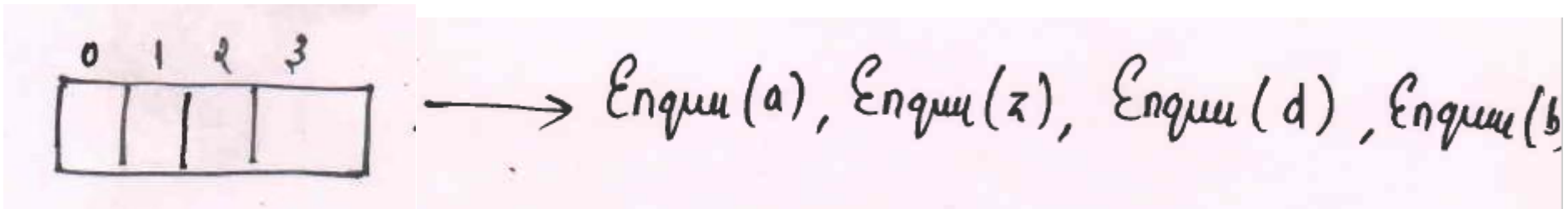


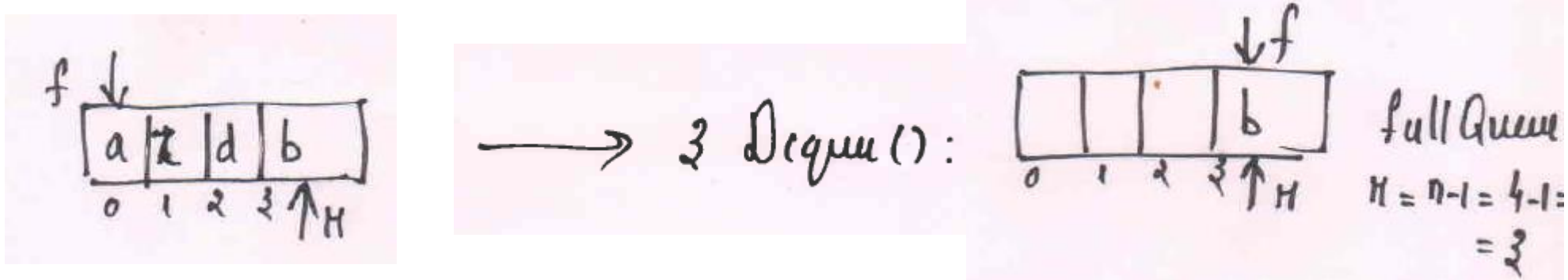
```
Dequeue () {
  if (IsEmpty ())
      return "underflow";
  else if (front == rear)
      front = rear = -1
  else {
      front = front+1
    }
}
```

getFront()



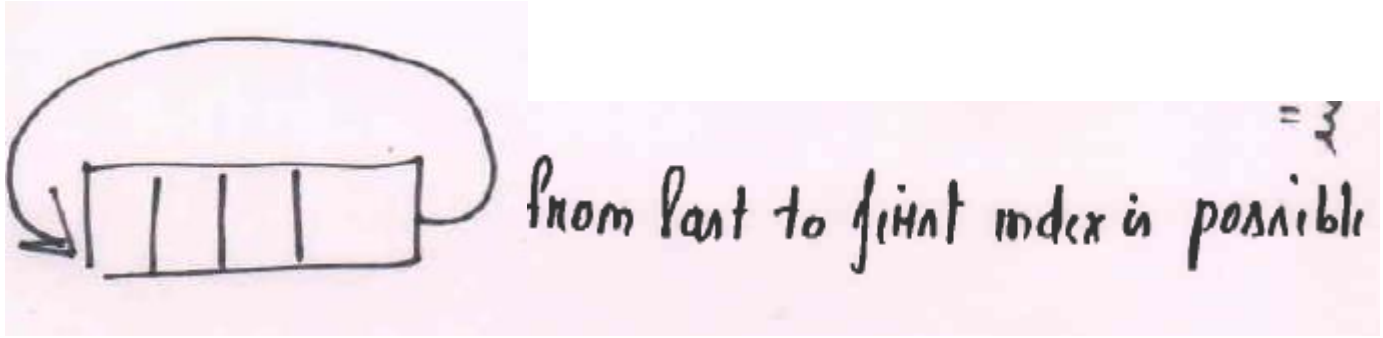
Limitation of Queue:





In Normal Queue Space is not utilised efficiently.

Circular Queue: Last position is connected back to the first position to make a circle.



- Current index is i, next index is (i+1)%n

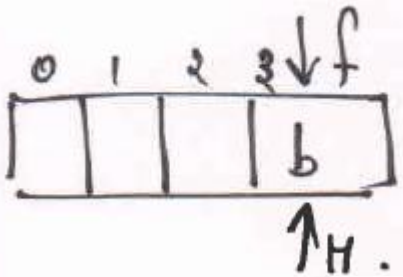
Circular queue

$i = 0$, next index is $(i+1) \% N = (0+1) \% 4 = 1 \% 4 = 1$

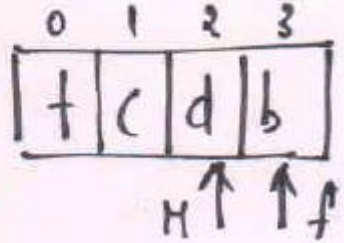
$i = 1$, next index is $(i+1) \% N = 2 \% 4 = 2$

$i = 2$, next index is $(2+1) \% 4 = 3 \% 4 = 3$

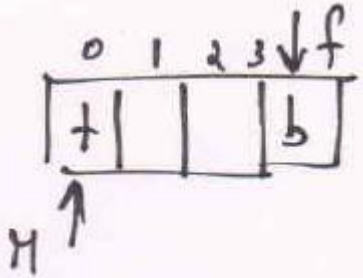
$i = 3$, next index $(3+1) \% 4 = 4 \% 4 = 0$ $\longrightarrow$ $i = n-1$, we have a change.



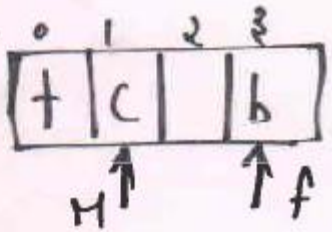
Επιλογή (d)



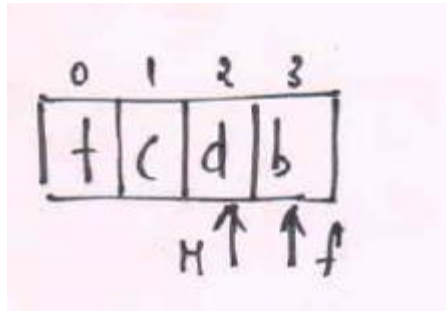
Επιλογή (t)



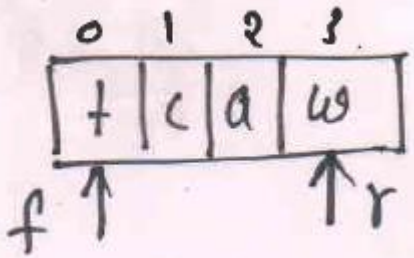
Επιλογή (c)



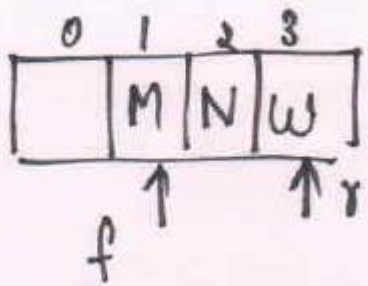
Enqueue (y)



full Queue = Head+1 = front . X



$\Longrightarrow (Head+1) \,\%\, N == front$



$(3+1) \,\%\, 4 = 0 \neq front = 1$

Circular Queue Vs Normal Queue

```
Enqueue (x) {
  if ((rear+1) %. N == front) {
      "Queue is full"
  }
  else if (IsEmpty()) {
     front = rear = 0;
     Queue [rear] = x;
  }
  else {
     rear = (rear+1) %. N
     Queue [rear] = x
  }
}
```

```
Enqueue (x) {
  if (rear == n-1)
      "Queue is full"
  else if (IsEmpty()) {
     front = rear = 0;
     Queue [rear] = x;
  }
  else {
     rear = rear+1
     Queue [rear] = x;
  }
}
```

Circular Queue Vs Normal Queue

```
Dequeue(){
  if(IsEmpty()){
      "Queue is Empty".
  }
  else if(front==rear){
      front=rear=-1;
  }
  else{
   front=(front+1)%N
  }
}
```

```
Dequeue(){
  if(IsEmpty())
      return "underflow".
  else if(front==rear)
      front=rear=-1
  else{
      front=front+1
  }
}
```

Circular Queue: Algorithms

```
Enqueue (x) {
  if ((rear+1) %. N == front) {
      "Queue is full"
  }
  else if (IsEmpty()) {
      front = rear = 0;
      Que[rear] = x;
  }
  else {
      rear = (rear+1) %. N
      Que[rear] = x
  }
}
```

```
Dequeue () {
  if (IsEmpty()) {
      "Queue is Empty".
  }
  else if (front == rear) {
      front = rear = -1;
  }
  else {
    front = (front+1) %. N
  }
}
```

```
Boolean IsEmpty () {

    if (front == -1 && rear == -1)
        return true;
    else
        return false;
}
```

```
getfront () {
    if (IsEmpty ())
        "Queue is Empty".
    else
return Queue [front];
}
```