01418231
Data Structures

LECTURE-4-QUEUE AND APPLICATIONS

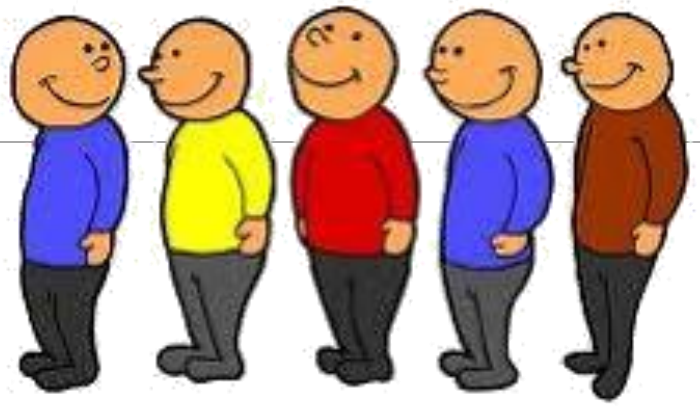
Powered by
Dr. Jirawan Charoensuk

Reference: picture

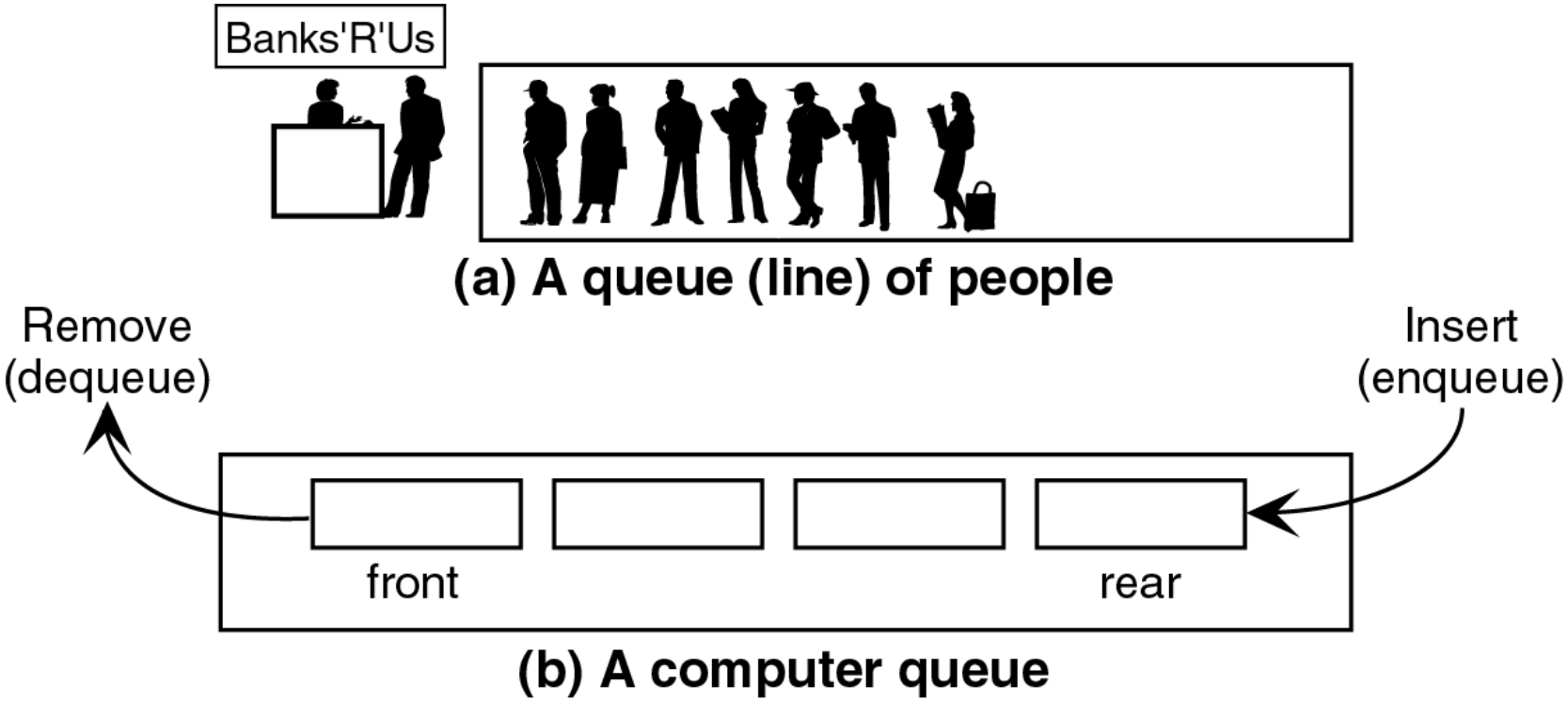
Agenda

- What is Queue ?
- Types of Queues
- What is the properties of queue ?
- Circular Queue
- The examples of applications uses queue concept
- Summary

What is Queue ?



Example of Queue

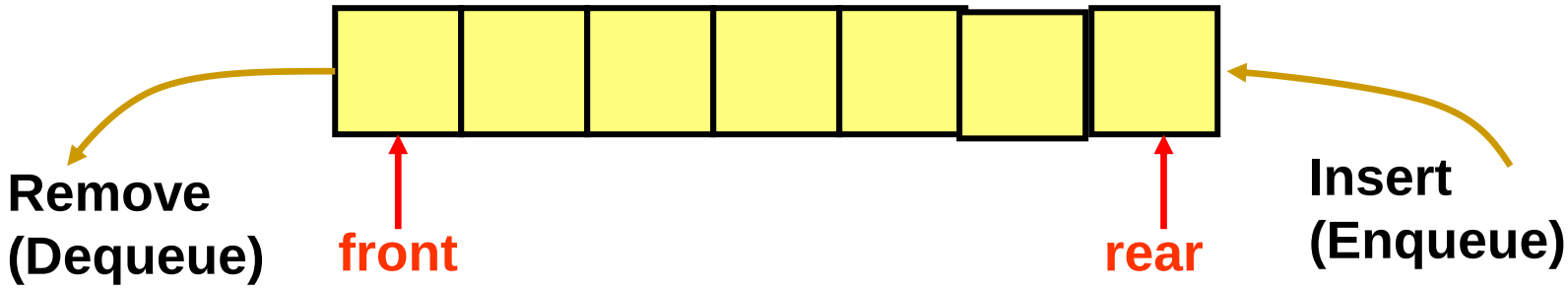


Queue

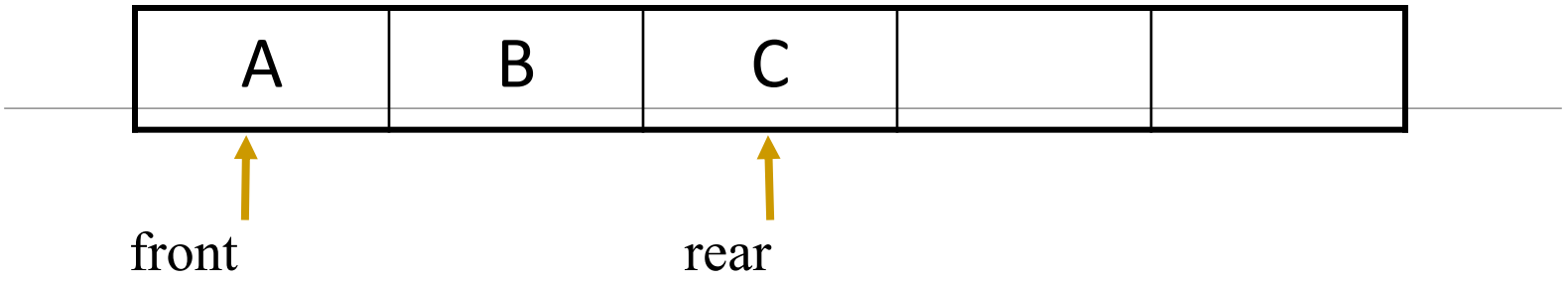
A queue is a list structure in which

- Inserted objects at the end of list, called "Enqueue"
- Deleted objects at the fist of list, called "Dequeue"

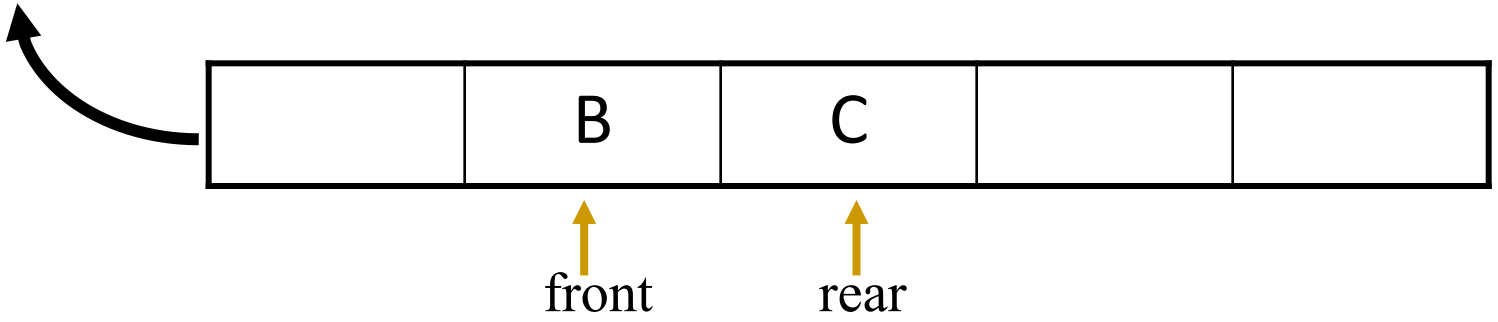
The policy is called First in, First out (FIFO)



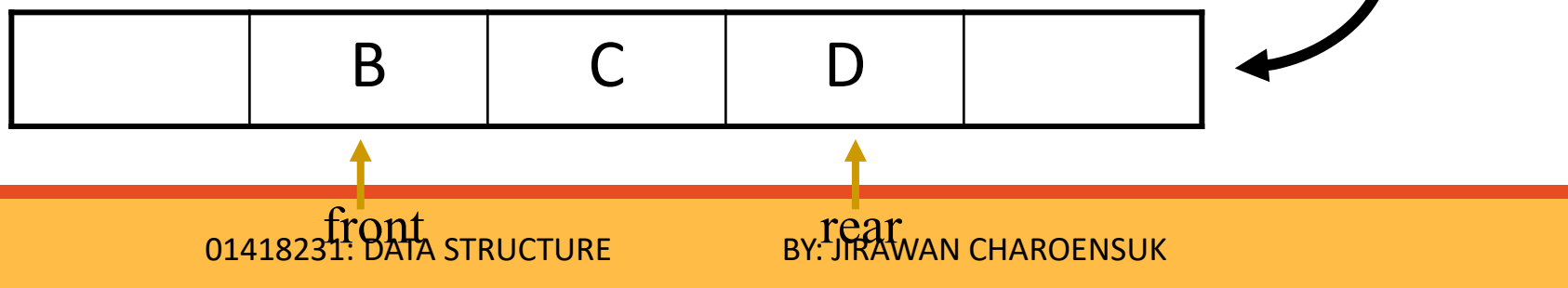
Queue



Dequeue



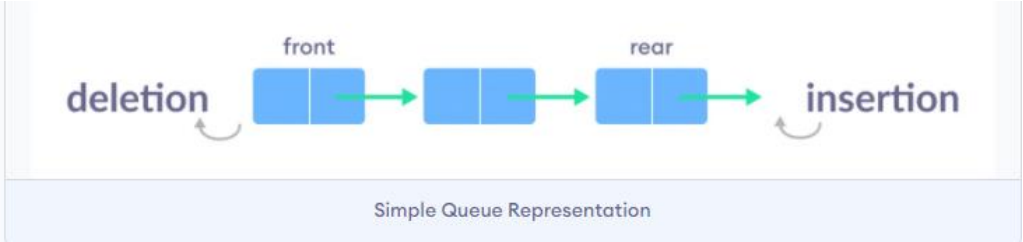
Enqueue



Types of Queues

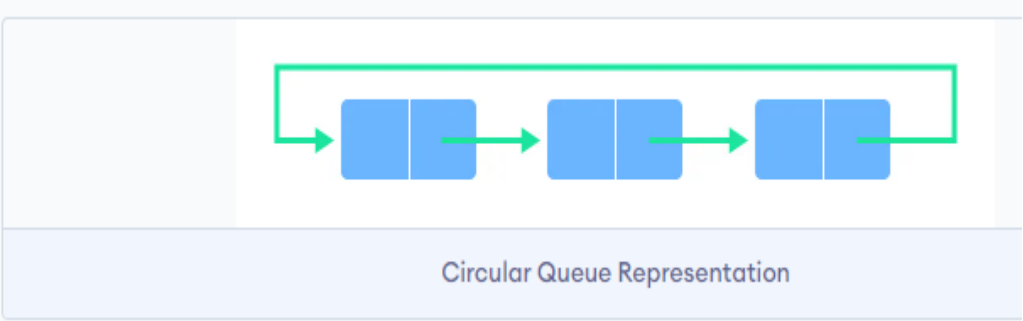
There are four different types of queues:

1. _____



Simple Queue Representation

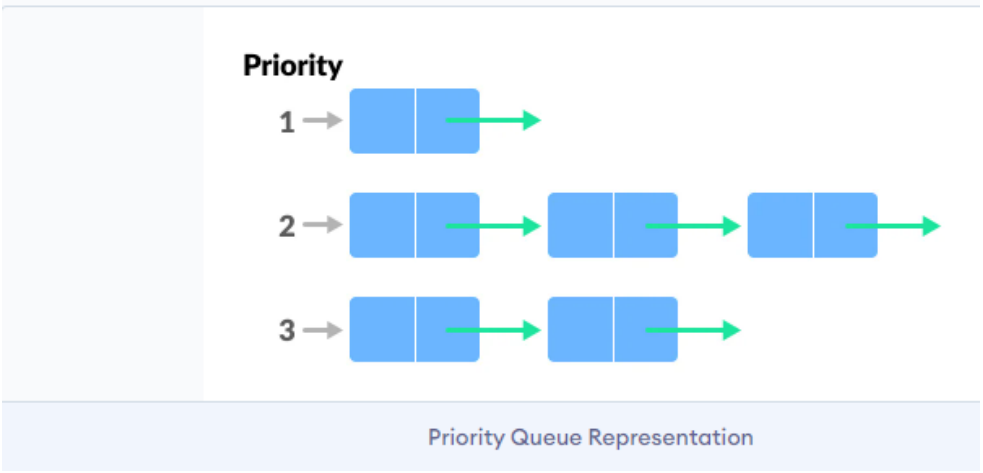
2. _____


Circular Queue Representation

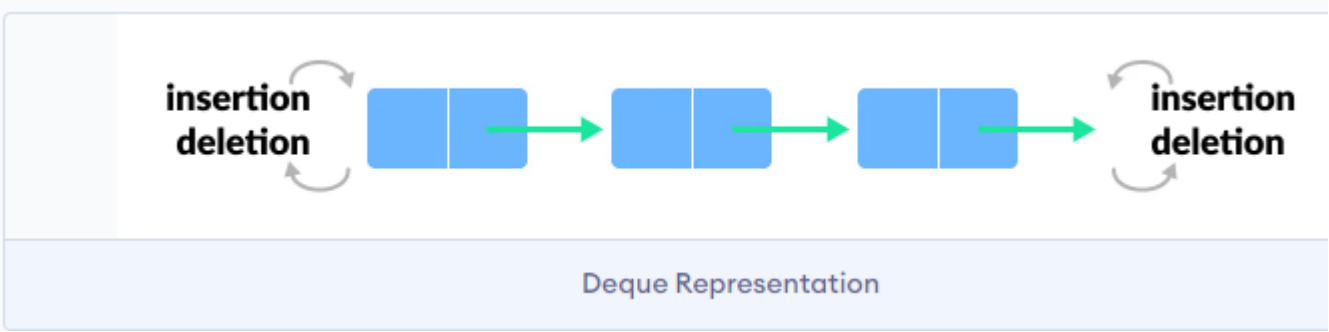
https://www.programiz.com/dsa/types-of-queue

Types of Queues

There are four different types of queues:



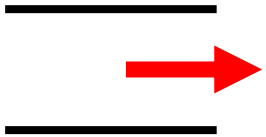
In a double ended queue, insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow the FIFO (First In First Out) rule.



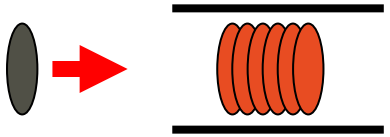
https://www.programiz.com/dsa/types-of-queue

Basic operations of queue

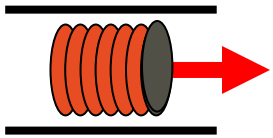
New



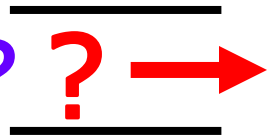
Enqueue



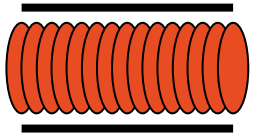
Dequeue



Empty? ?



Full?



Queue's Operations

Idea: a "First in, First out" (FIFO) data structure

Function:

- _____ -> Inserts object o at the rear of the queue
- _____ -> Removes the object from the front of the queue; an error occurs if the queue is empty
- _____ -> Returns the position of beginning of queue, an error occurs if the queue is empty
- _____ Returns the position of ending of queue, an error occurs if the queue is empty

Queue's Operations

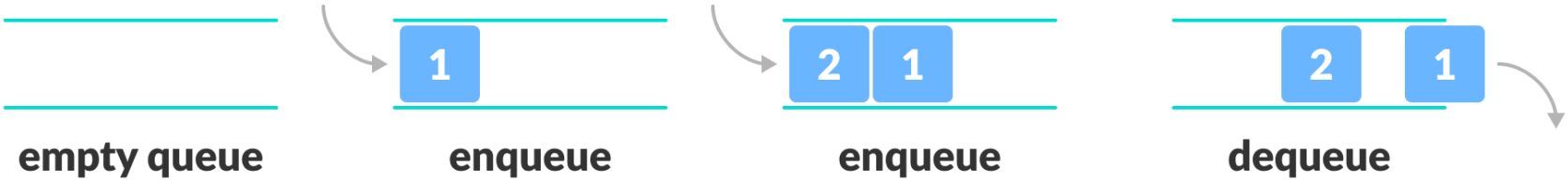
Function:

- New(S) -> Creates an empty queue
- Size(S) -> Size of queue (integer)
- MaxQueue(S) -> Maximum of queue (integer)
- IsEmpty(S) -> Is queue empty? (boolean)
- IsFull(S) -> Is queue full? (boolean)

Array implementation

Static
- Array

Dynamic
- Linked-list



https://www.programiz.com/dsa/queue

Array-based Queue

Queue is represented as a partially filled array

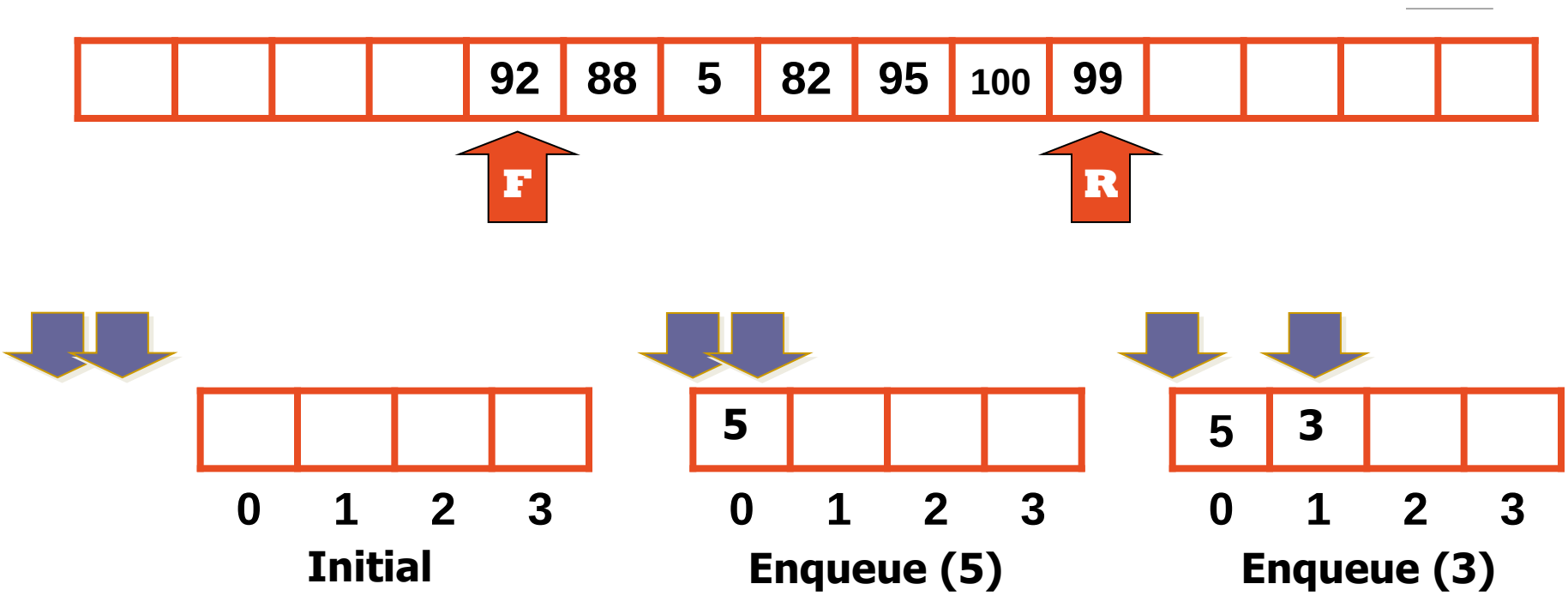
Use an array of size N

Two variables keep track of the front and rear

f index of the front element

r index immediately past the rear element

Array-based Queue



Queue's Operations

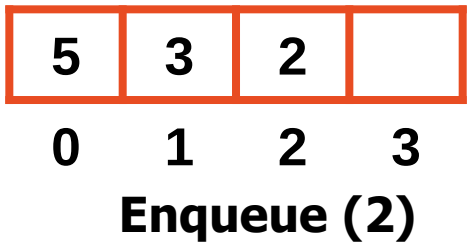
If Queue is full or Rear >= MaxQueue(S) ,we can't Enqueue(S)

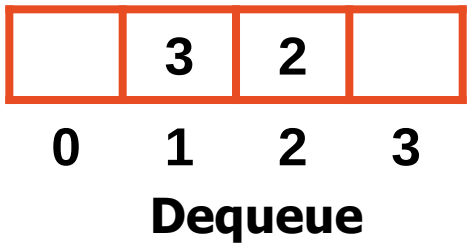
- Because Queue is Overflow

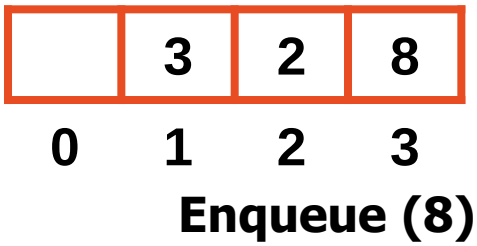
If Queue is empty, we can't Dequeue(S)

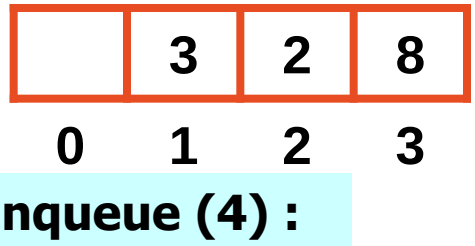
- Because Queue is Underflow

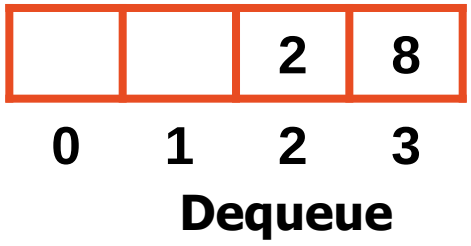
Queue's Operations

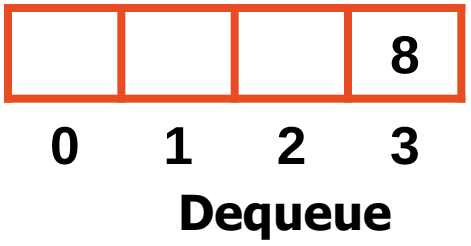


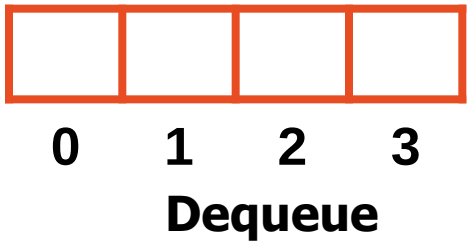


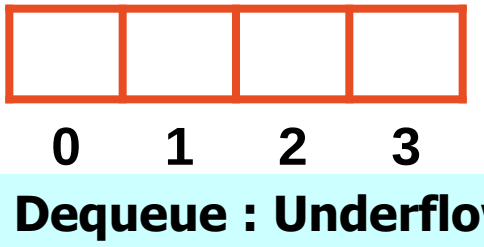




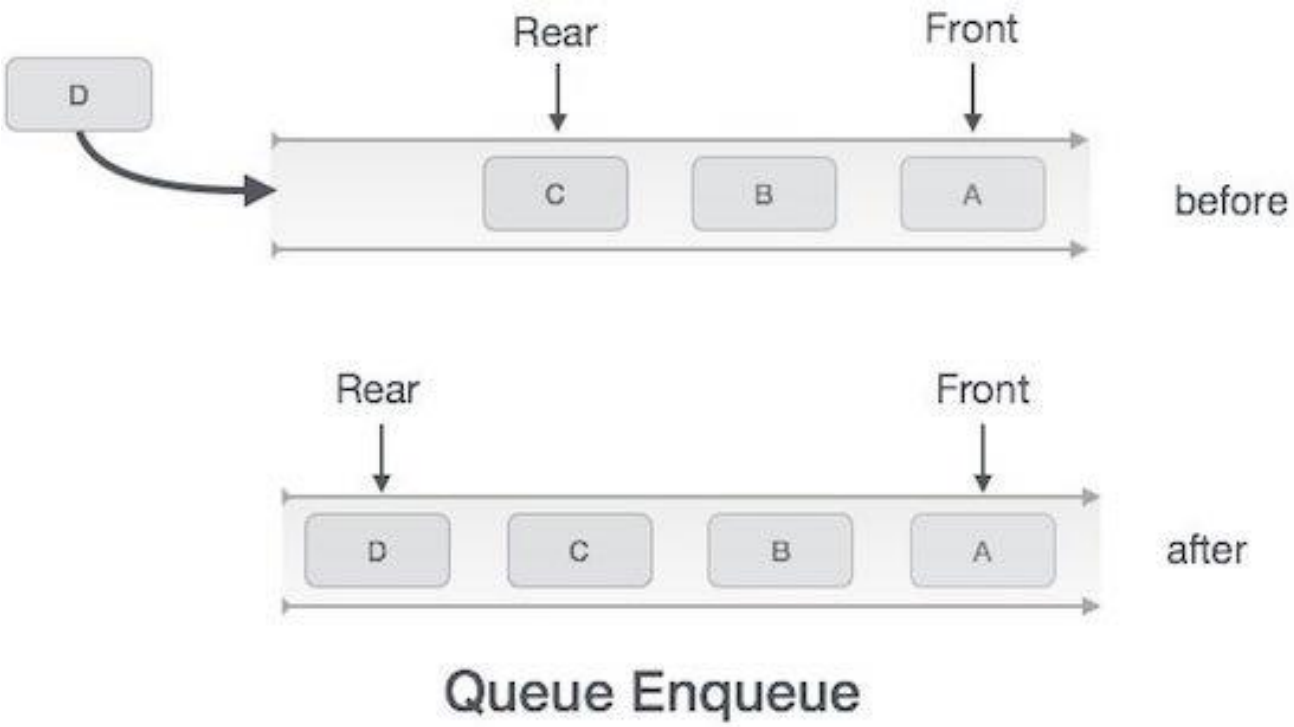






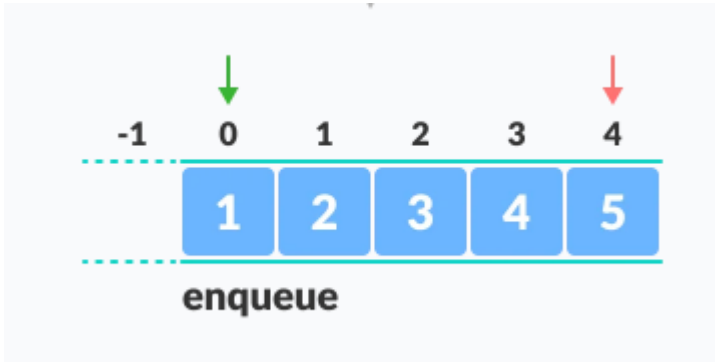


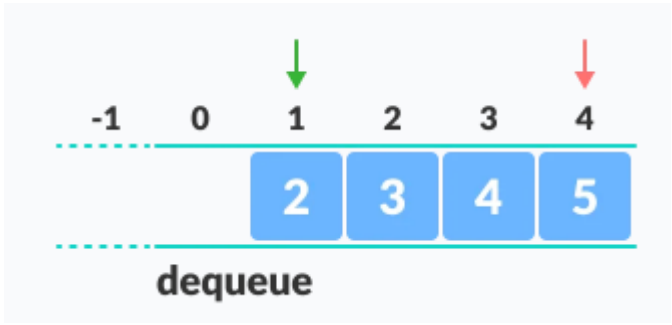
Enqueue Operation

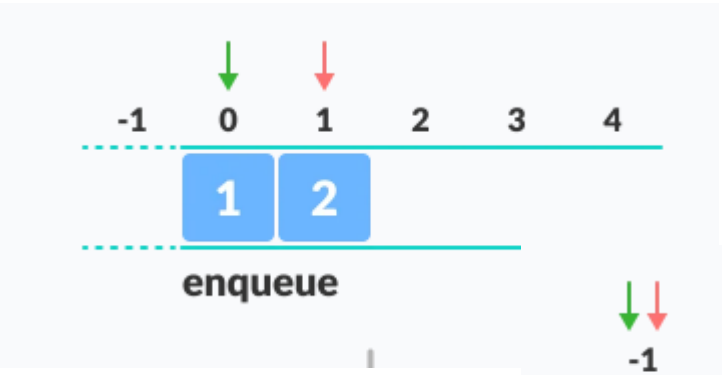


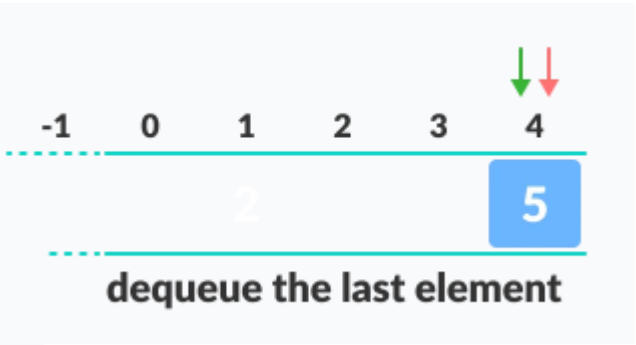
Queue Enqueue

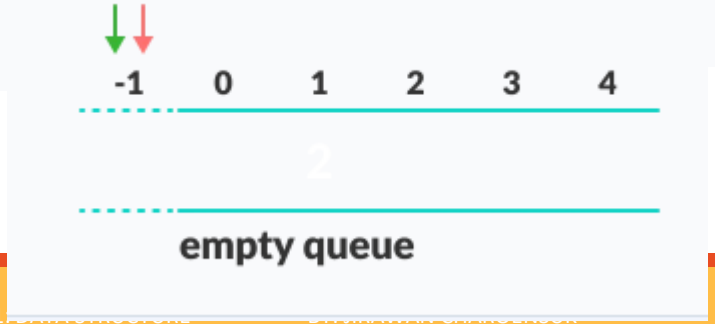
https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm











https://www.programiz.com/dsa/queue

Queue Operations using Array

Enqueue Operation Queues maintain two data pointers, front and rear.

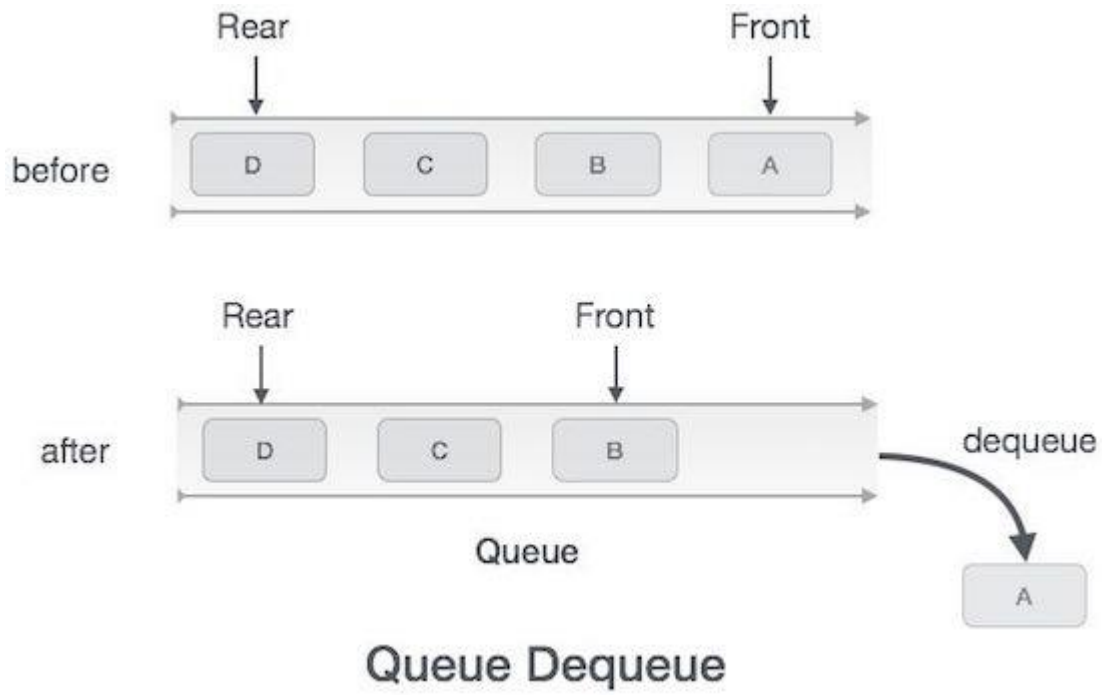
- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, print “Queue is Full”
- **Step 3** − If the queue is not full and check the first element of queue, set front = 0
- **Step 4** − If the queue is not full and not the first element of queue, increment **rear** pointer to point the next empty space.
- **Step 5** − Add data element to the queue location, where the rear is pointing and print value

```
1.  void enQueue(int value) {
2.    if (rear == SIZE - 1)
3.      printf("\nQueue is Full!!");
4.    else {
5.      if (front == -1)
6.        front = 0;
7.      rear++;
8.      items[rear] = value;
9.      printf("\nInserted -> %d", value);
10.   }
11. }
```

https://www.programiz.com/dsa/queue

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

Dequeue Operation



Queue Dequeue

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

Queue Operations using Array

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.

- **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.

.

```
1.  void deQueue() {
2.    if (front == -1)
3.      printf("\nQueue is Empty!!");
4.    else {
5.      printf("\nDeleted : %d",
    items[front]);
6.      front++;
7.      if (front > rear)
8.        front = rear = -1;
9.    }
10. }
```

https://www.programiz.com/dsa/queue

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

```
1.  //https://www.programiz.com/dsa/queue
2.  #include <stdio.h>
3.  #include <stdlib.h>

4.  #define SIZE 5

5.  void enQueue(int);
6.  void deQueue();
7.  void display();

8.  int items[SIZE], front = -1, rear = -1;

9.  main() {
10.   deQueue();
11.   enQueue(1);display();
12.   enQueue(2);display();
13.   enQueue(3);display();
14.   enQueue(4);display();
15.   enQueue(5);display();
16.   enQueue(6);display();
17.   deQueue();display();
18. }
```

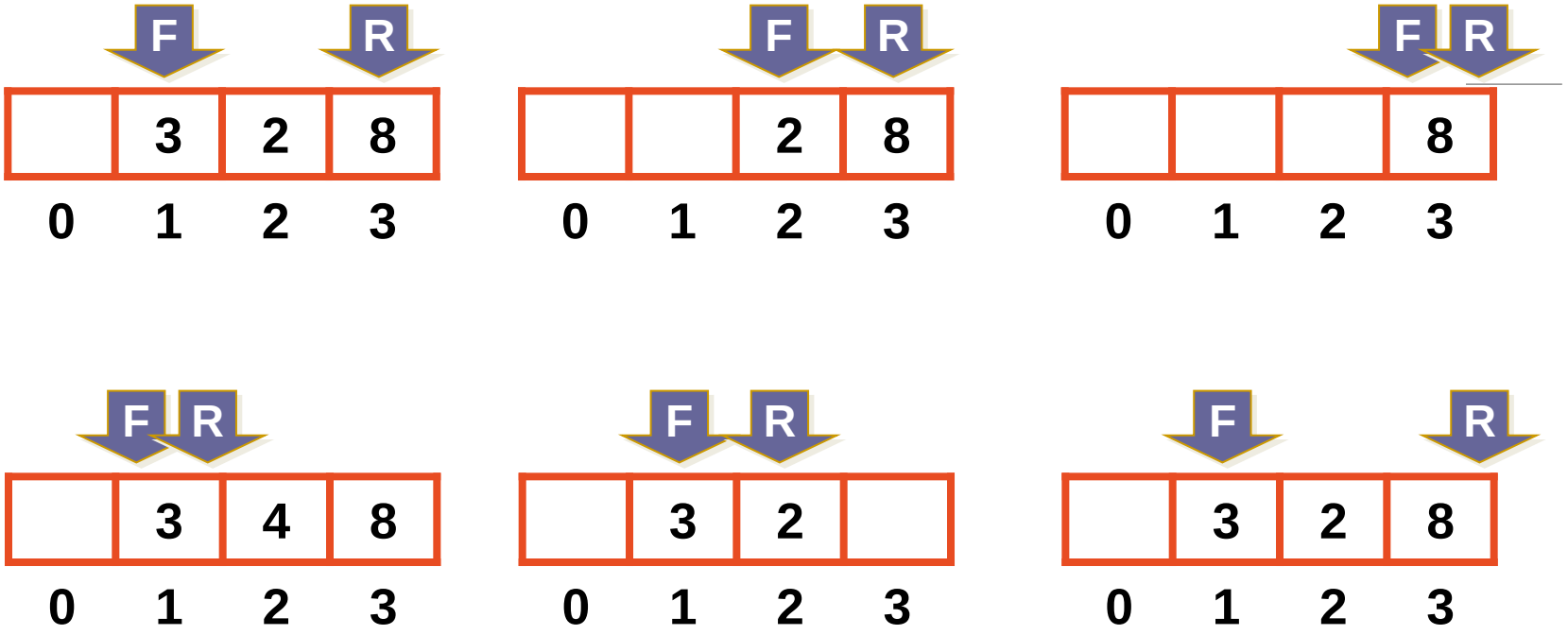
```
19. void enQueue(int value) {
20.   if (rear == SIZE - 1)
21.     printf("\nQueue is Full!!");
22.   else {
23.     if (front == -1)
24.       front = 0;
25.     rear++;
26.     items[rear] = value;
27.     printf("\nInserted -> %d", value);
28.   }
29. }

30. void deQueue() {
31.   if (front == -1)
32.     printf("\nQueue is Empty!!");
33.   else {
34.     printf("\nDeleted : %d", items[front]);
35.     front++;
36.     if (front > rear)
37.       front = rear = -1;
38.   }
39. }
```

https://www.programiz.com/dsa/queue

```
40. void display() {
41.   if (rear == -1)
42.     printf("\nQueue is Empty!!!");
43.   else {
44.     int i;
45.     printf("\nQueue elements are:\n [ ");
46.     for (i = front; i <= rear; i++){
47.       printf("%d ", items[i]);
48.     }
49.   }
50.   printf("]\n");
51. }
```

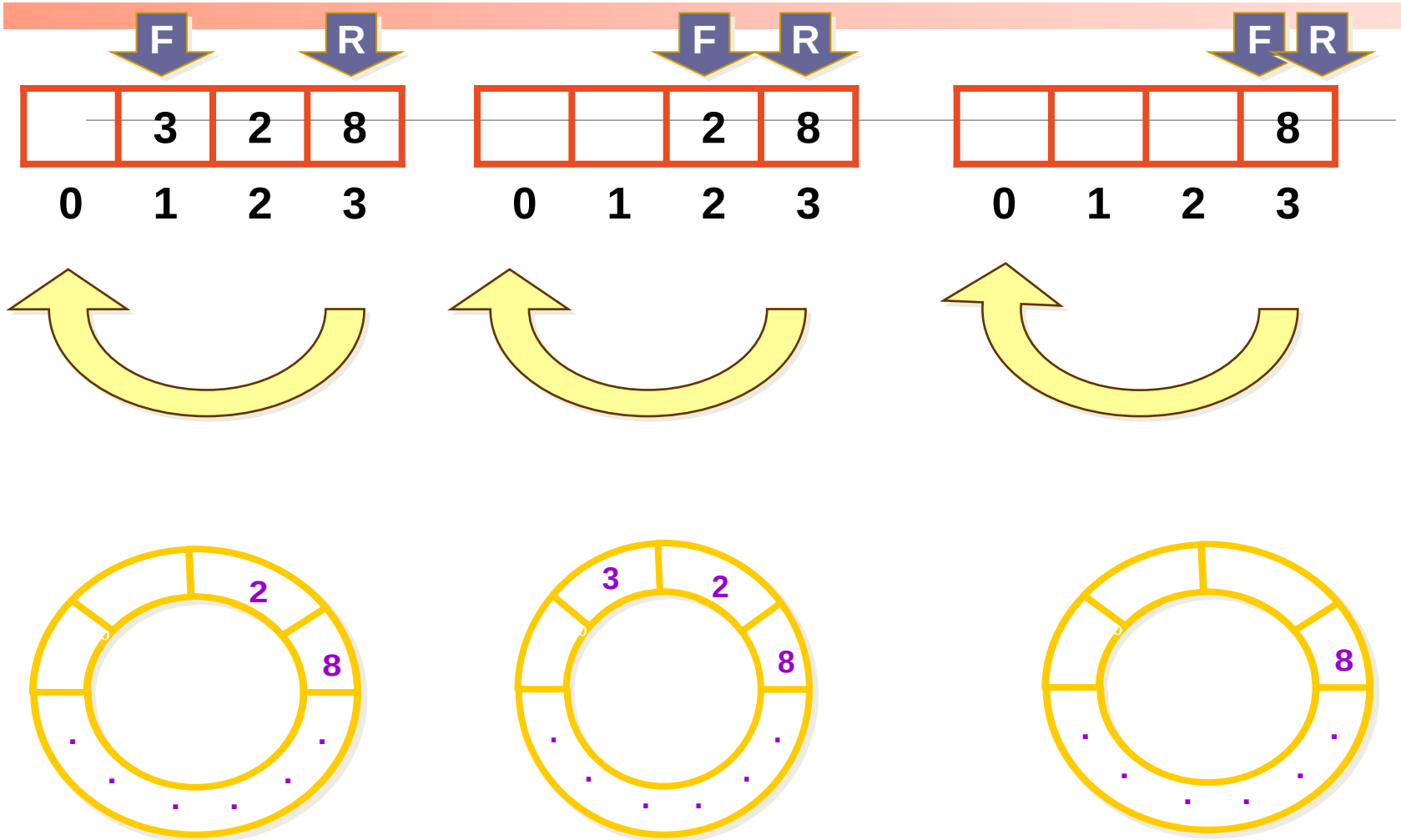
https://www.programiz.com/dsa/queue

Limited of Array-based Queue



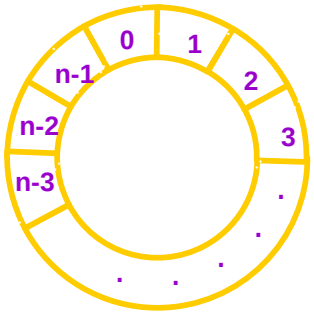
- We have free zone before front(f), we can't enqueue if [rear (r) = n-1] (Queue Fulled)

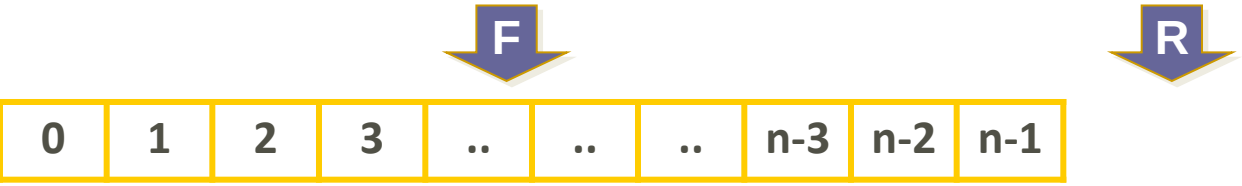
Solution :: Circular Queue



Circular Queue vs Linear Queue

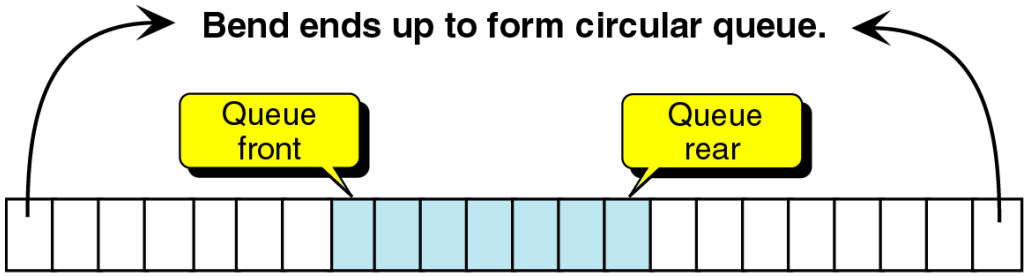
IF R = N-1 THEN
R = 0

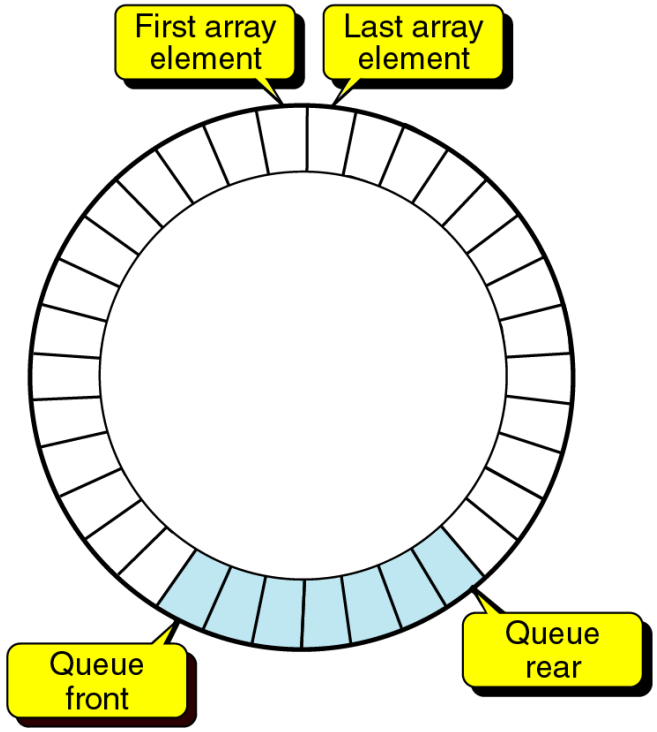




IF R = N-1 THEN
write "Queue is full"

Circular Queue



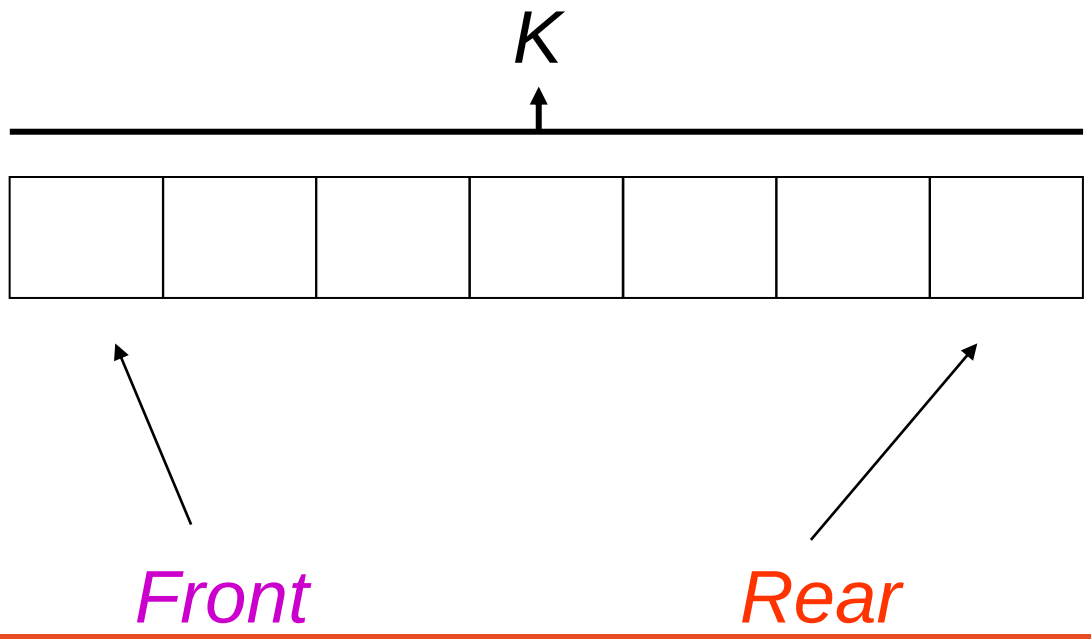


How to Advance

Again we use an array of fixed size K

But now we keep two indexes

- Front and Rear



How to Advance

Both front & back pointers should make advancement until they reach end of the array.
should re-point to beginning of the array

Alternatively, use modular arithmetic:

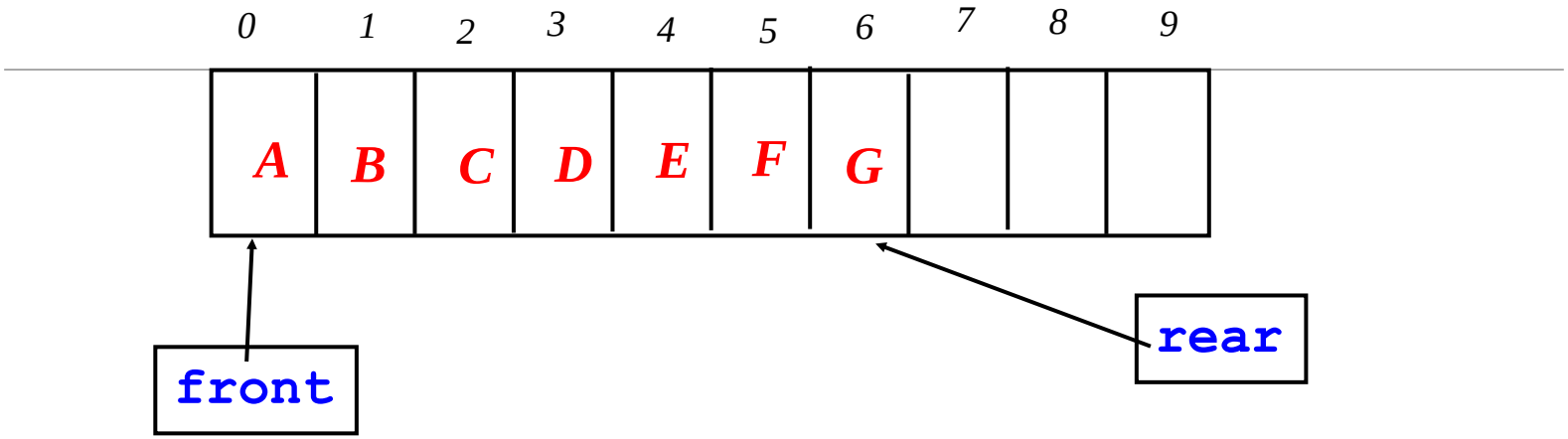
```
public static int rear)
  { return ((rear+1) % maxsize);
  }
```

mod operator

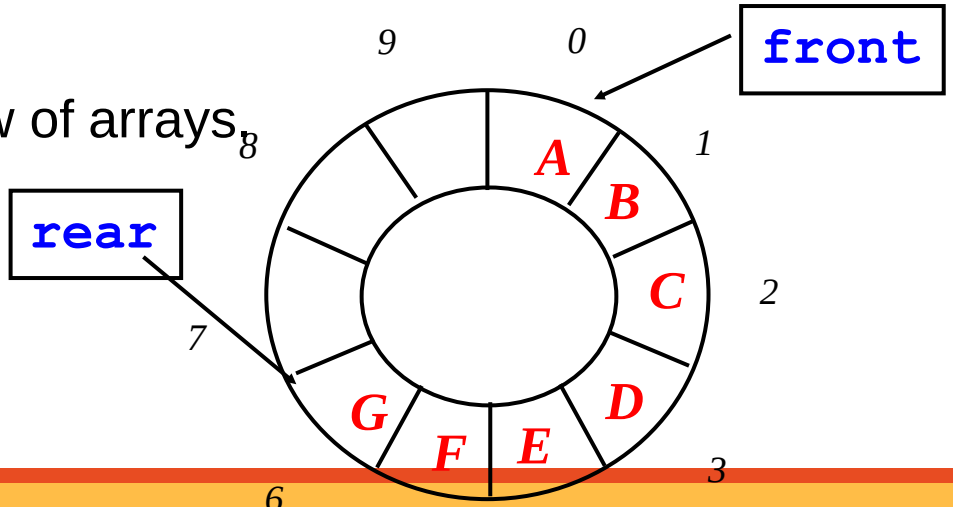
```
public static int rear)
  { int rear = rear+1;
    if (rear<=maxsize) return r;
    else return 0;
  }
```

upper bound of the array

Circular queue

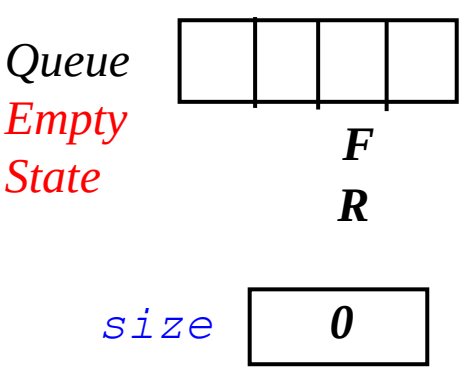


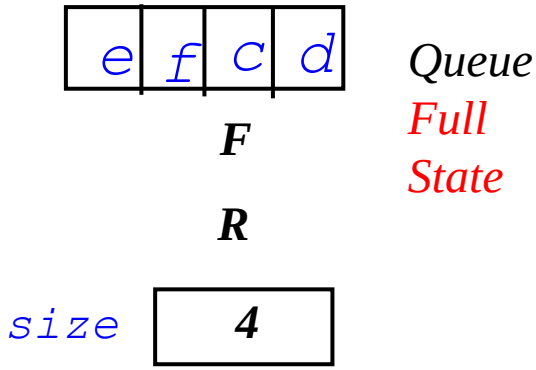
Circular view of arrays



Checking for Full/Empty State

What does (F == R) denote?

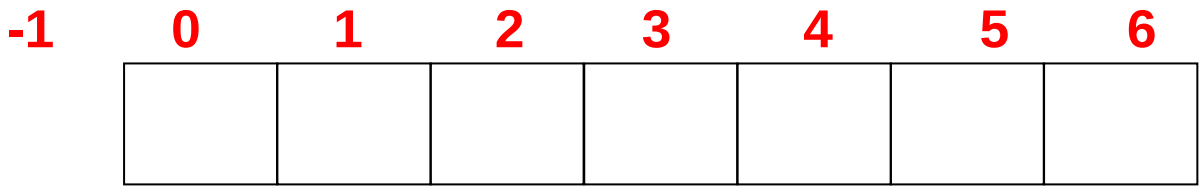




New Queue (Array)

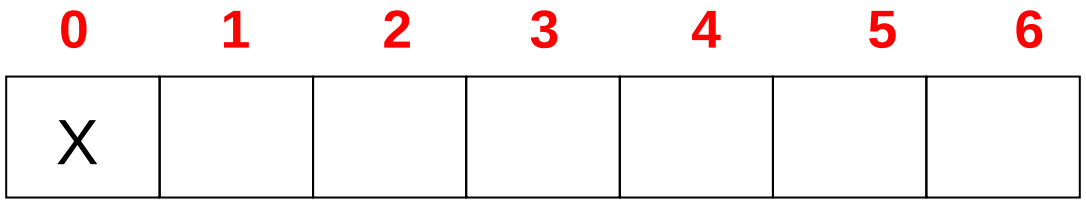
Front = -1

Rear = -1



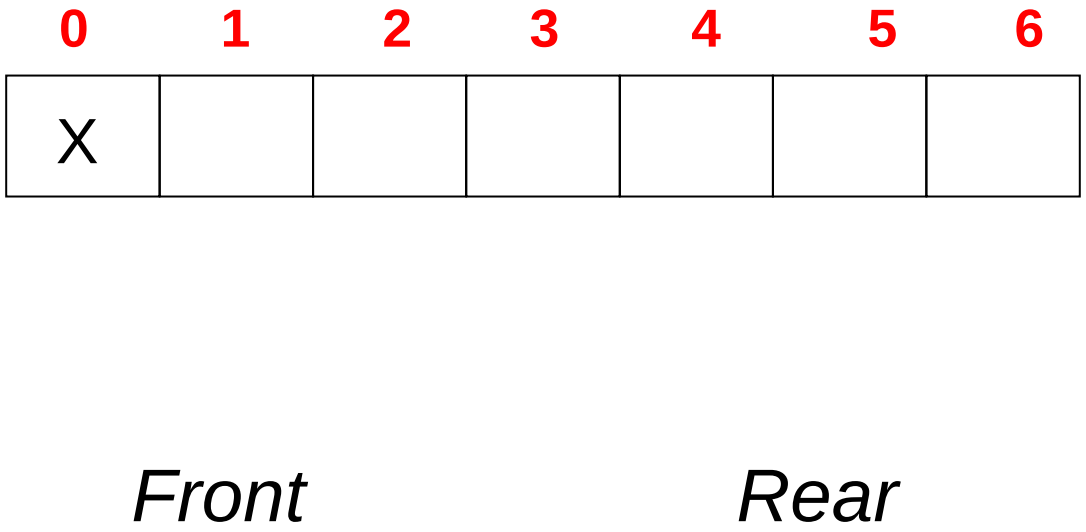
Enqueue(x)

Put x into array at Rear, then move Rear



Enqueue(y), Enqueue(z), Enqueue(w)

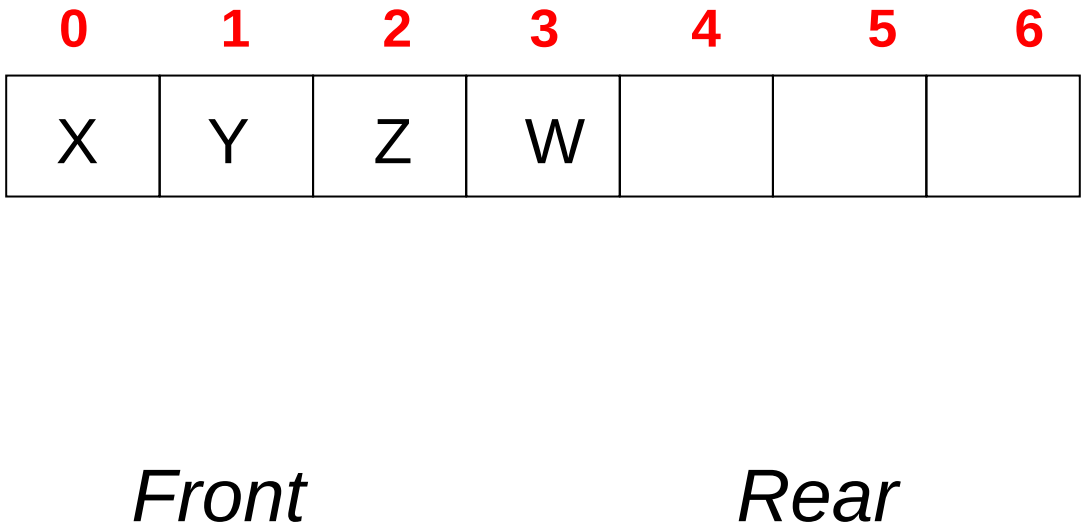
How to put y,z,w into queue ?



Dequeue

Return element x at *Front*

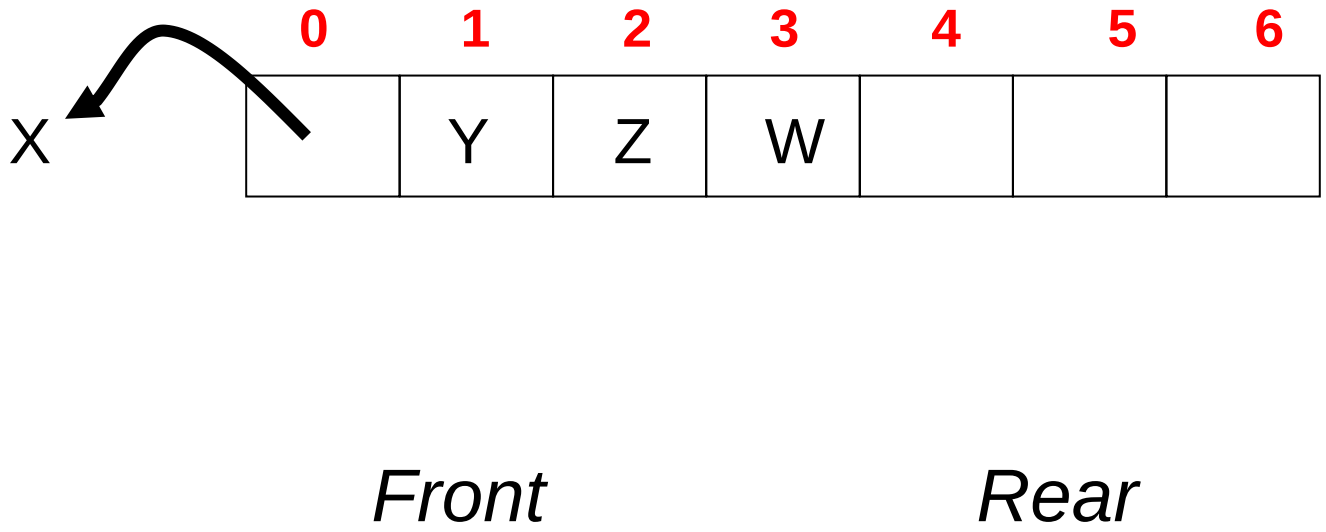
Advance Front



Dequeue

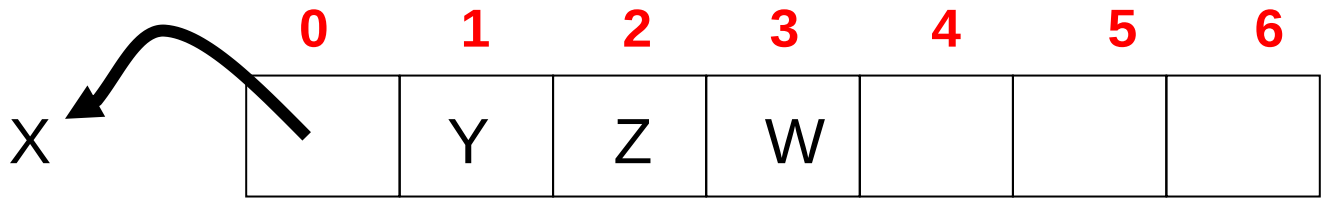
Return element x at *Front*

Advance Front



Dequeue

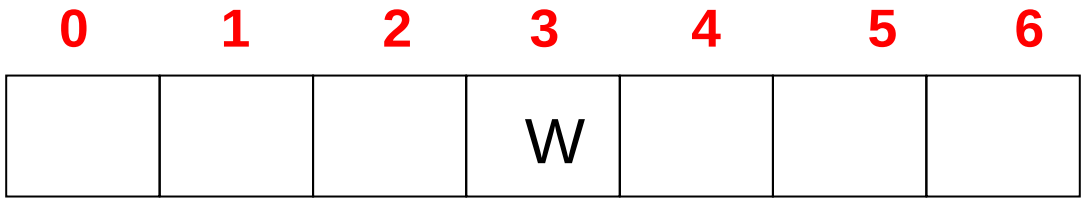
How to remove element y,z from queue ?



Front *Rear*

Enqueue(y), Enqueue(z)

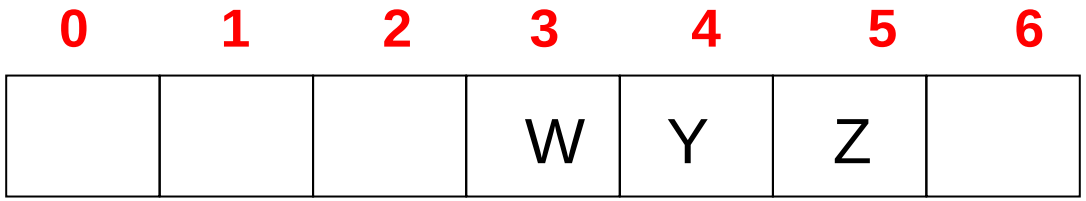
How to put y,z into queue ?



Front = ? *Rear = ?*

Dequeue

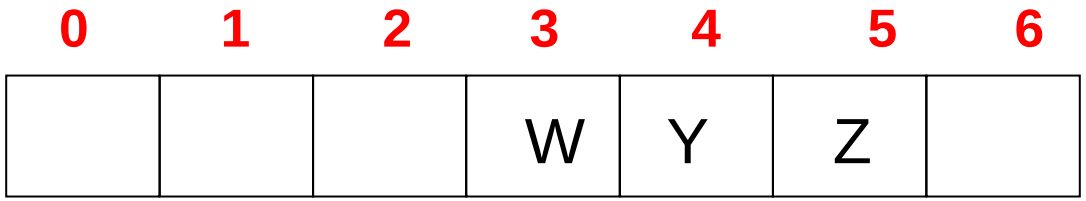
How to remove element w from queue ?



Front = ? *Rear = ?*

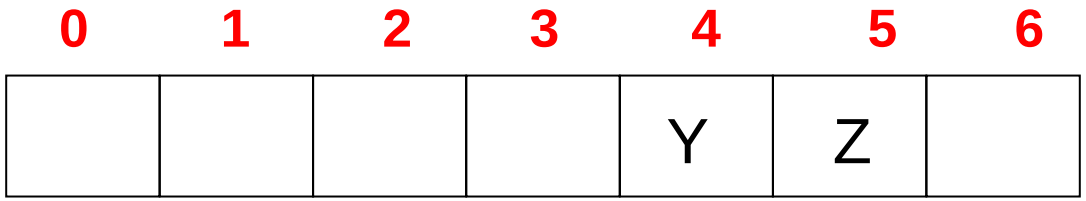
Dequeue

How to remove element w from queue ?



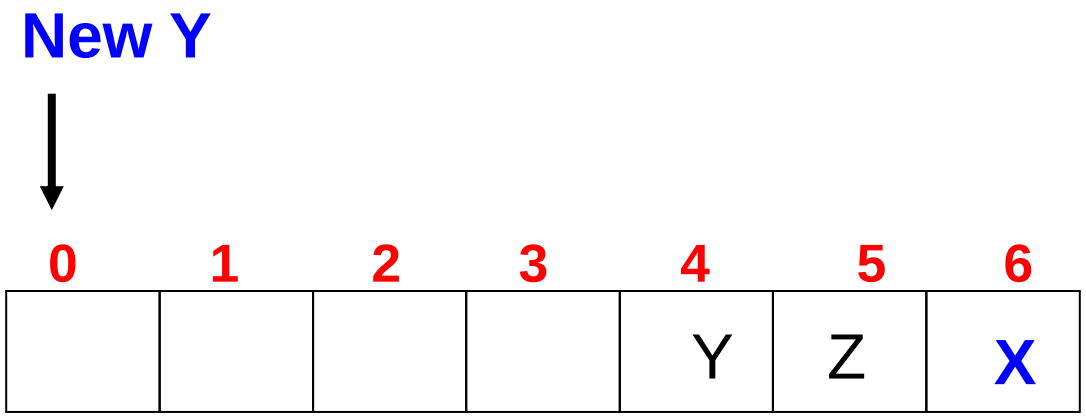
Dequeue

How to remove element w from queue ?



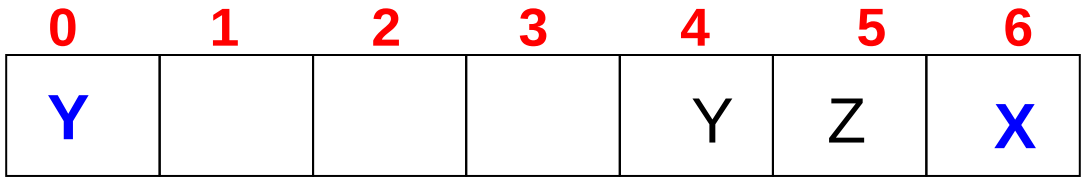
Enqueue(x)

But what happens when Rear is at end of array



Enqueue(x)

But what happens when Rear is at end of array

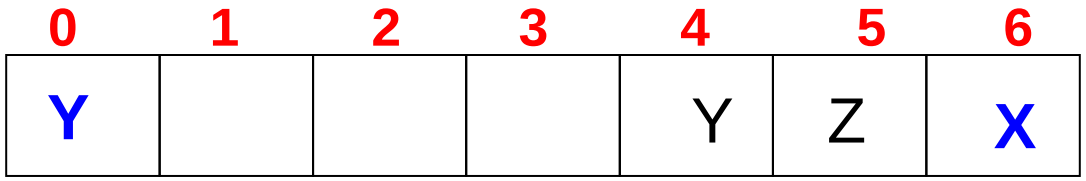


Front *Rear =?*

Enqueue(x)

But what happens when Rear is at end of array

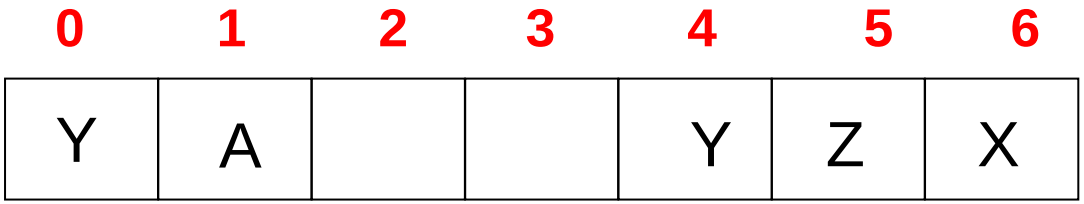
Rear =



Front *Rear*

Enqueue(A)

Rear =

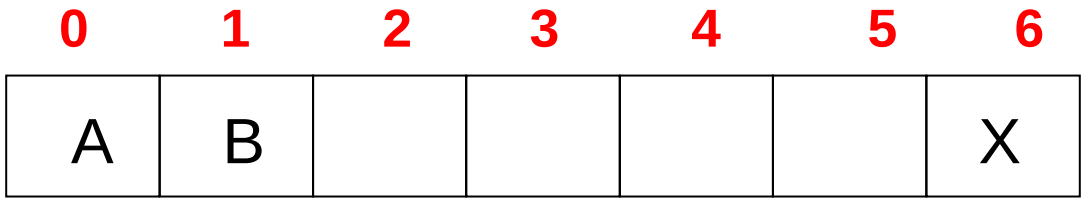


Front *Rear*

Dequeue

And what happens when Front reaches end?

Same thing: Front ← (Front + 1) modulo K

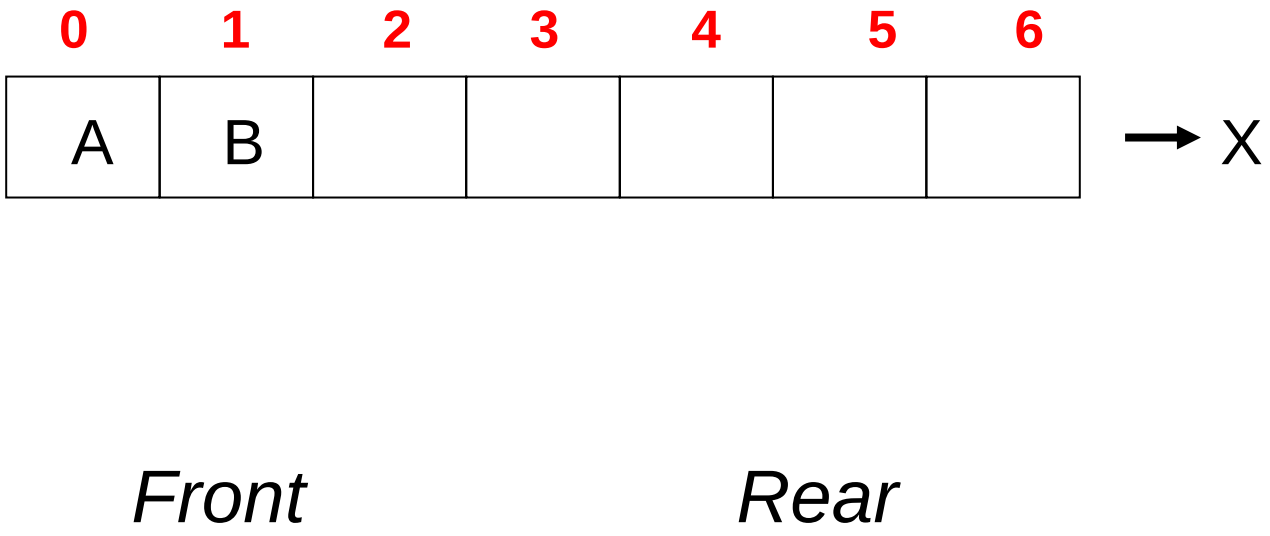


Front *Rear*

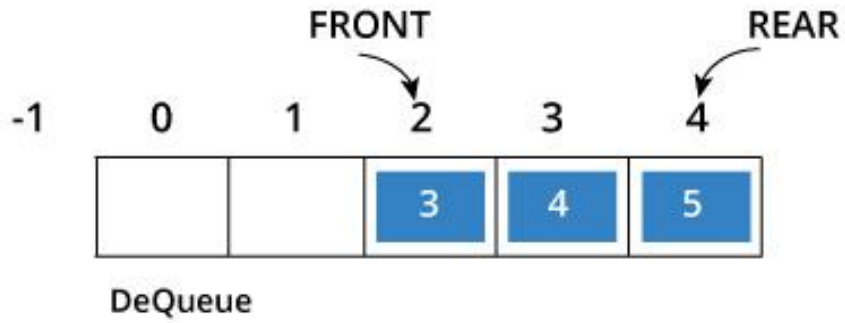
Dequeue

And what happens when Front reaches end?

Same thing: Front ← (Front + 1) % K



Circular Queue



https://www.programiz.com/dsa/circular-queue

Queue operations

Two pointers called FRONT and REAR are used to keep track of the first and last elements in the queue.

- When initializing the queue, we set the value of FRONT and REAR to -1.
- On enqueing an element, we circularly increase the value of REAR index and place the new element in the position pointed to by REAR.
- On dequeueing an element, we return the value pointed to by FRONT and circularly increase the FRONT index.

- Before enqueing, we check if queue is already full.
- Before dequeuing, we check if queue is already empty.
- When enqueing the first element, we set the value of FRONT to 0.
- When dequeing the last element, we reset the values of FRONT and REAR to -1.

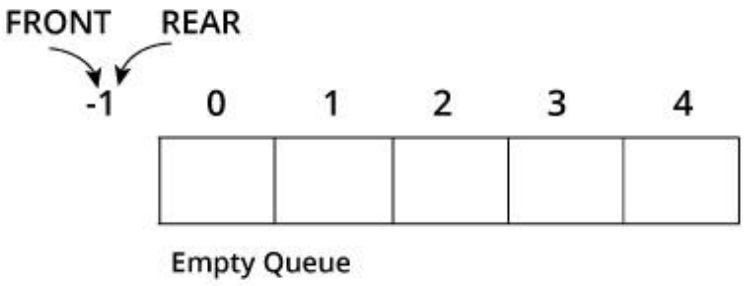
Queue operations

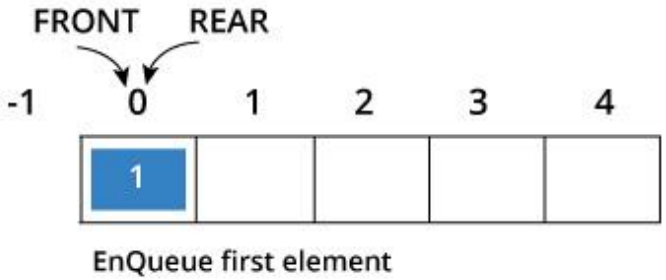
However, the check for full queue has a new additional case:

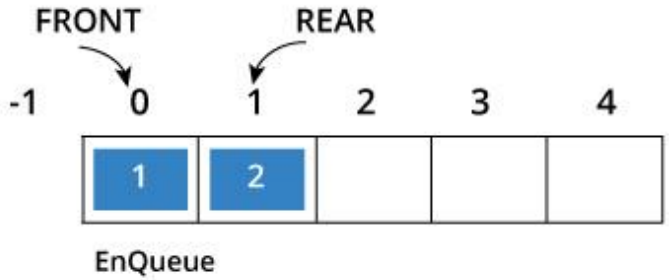
Case 1: FRONT = 0 && REAR == SIZE - 1

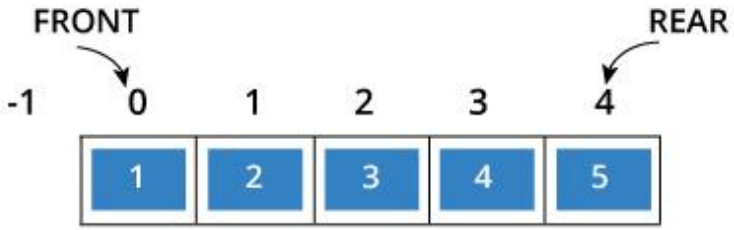
Case 2: FRONT = REAR + 1

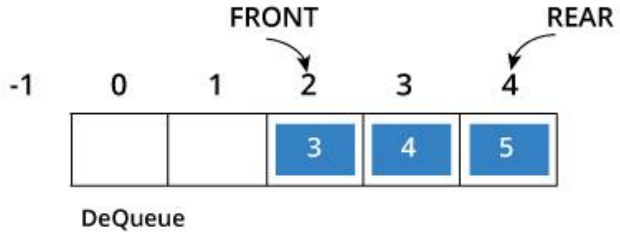
- The second case happens when REAR starts from 0 due to circular increment and when its value is just 1 less than FRONT, the queue is full.

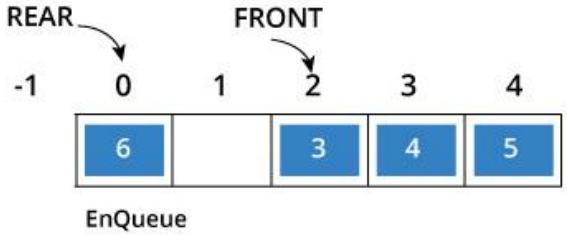

FRONT
REAR
-1
0
1
2
3
4
Empty Queue

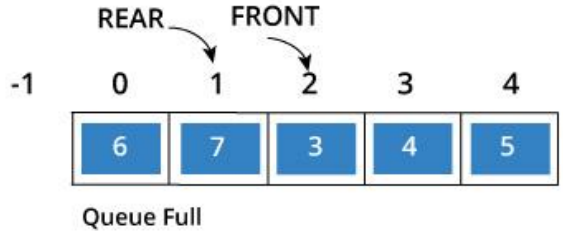

FRONT
REAR
-1
0
1
2
3
4
1
EnQueue first element


FRONT
REAR
-1
0
1
2
3
4
1
2
EnQueue


FRONT
REAR
-1
0
1
2
3
4
1
2
3
4
5


FRONT
REAR
-1
0
1
2
3
4
3
4
5
DeQueue


REAR
FRONT
-1
0
1
2
3
4
6
3
4
5
EnQueue


REAR
FRONT
-1
0
1
2
3
4
6
7
3
4
5
Queue Full

```
1.  // Circular Queue implementation in C
2.  //https://www.programiz.com/dsa/circular-queue
3.  #include <stdio.h>
4.  #include <stdlib.h>
5.  #define SIZE 5

6.  int items[SIZE];
7.  int front = -1, rear = -1;

8.  // Check if the queue is full
9.  int isFull() {
10.   if ((front == rear + 1) || (front == 0 && rear == SIZE - 1)) return 1;
11.   return 0;
12. }

13. // Check if the queue is empty
14. int isEmpty() {
15.   if (front == -1) return 1;
16.   return 0;
17. }
```

https://www.programiz.com/dsa/circular-queue

```
1.  // Adding an element
2.  void enQueue(int element) {
3.    if (isFull())
4.      printf("\n Queue is full!! \n");
5.    else {
6.      if (front == -1) front = 0;
7.      rear = (rear + 1) % SIZE;
8.      items[rear] = element;
9.      printf("\n Inserted -> %d",
    element);
10.   }
11. }
```

```
1.  // Removing an element
2.  int deQueue() {
3.    int element;
4.    if (isEmpty()) {
5.      printf("\n Queue is empty !! \n");
6.      return (-1);
7.    } else {
8.      element = items[front];
9.      if (front == rear) {
10.       front = -1;
11.       rear = -1;
12.     }
13.     // Q has only one element, so we reset the
14.     // queue after dequeing it. ?
15.     else {
16.       front = (front + 1) % SIZE;
17.     }
18.     printf("\n Deleted element -> %d \n", element);
19.     return (element);
20.   }
21. }
```

https://www.programiz.com/dsa/circular-queue

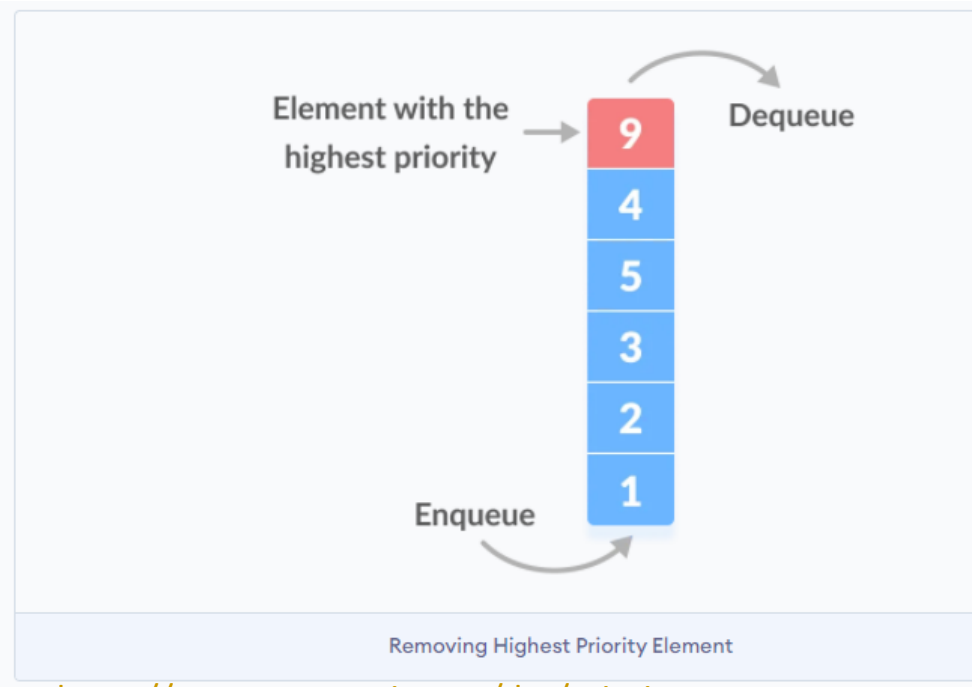
```
1.  // Display the queue
2.  void display() {
3.    int i;
4.    if (isEmpty())
5.      printf(" \n Empty Queue\n");
6.    else {
7.      printf("\n Front -> %d ", front);
8.      printf("\n Items -> ");
9.      for (i = front; i != rear; i = (i + 1) %
    SIZE) {
10.       printf("%d ", items[i]);
11.     }
12.     printf("%d ", items[i]);
13.     printf("\n Rear -> %d \n", rear);
14.   }
15. }
```

```
1.  main() {
2.    // Fails because front = -1
3.    deQueue();

4.    enQueue(1);display();
5.    enQueue(2);display();
6.    enQueue(3);display();
7.    enQueue(4);display();
8.    enQueue(5);display();

9.    // Fails to enqueue because front == 0 &&
    rear == SIZE - 1
10.   enQueue(6);display();
11.   enQueue(7);display();

12.   // Fails to enqueue because front == rear
    + 1
13.   enQueue(8);display();
14. }
```

https://www.programiz.com/dsa/circular-queue

Priority Queue

- A priority queue is a special type of queue in which each element is associated with a priority value. And, elements are served on the basis of their priority. That is, higher priority elements are served first.
- However, if elements with the same priority occur, they are served according to their order in the queue.



https://www.programiz.com/dsa/priority-queue

Double ended queue

- Deque or Double Ended Queue is a type of queue in which insertion and removal of elements can either be performed from the front or the rear. Thus, it does not follow FIFO rule (First In First Out).

https://www.programiz.com/dsa/deque

The examples of applications uses queue concept

Queue Applications

Queues are found whenever supply and demand (*servers* and *clients*) cannot be assured to stay in lockstep

Data may come in too quickly, and have to be held for processing later

- Communications Buffer
- Printer queues
- Time-shared computer system

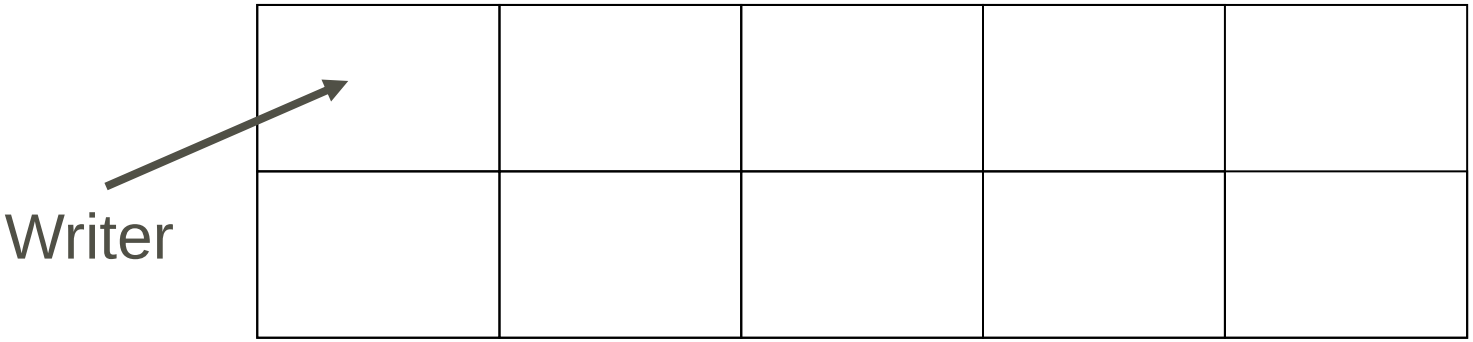
Communications Buffer

Two processes:

- Writer: Puts information out
- Reader: Gets information in

We want each process to be independent

- Asynchronous: Work at their own pace



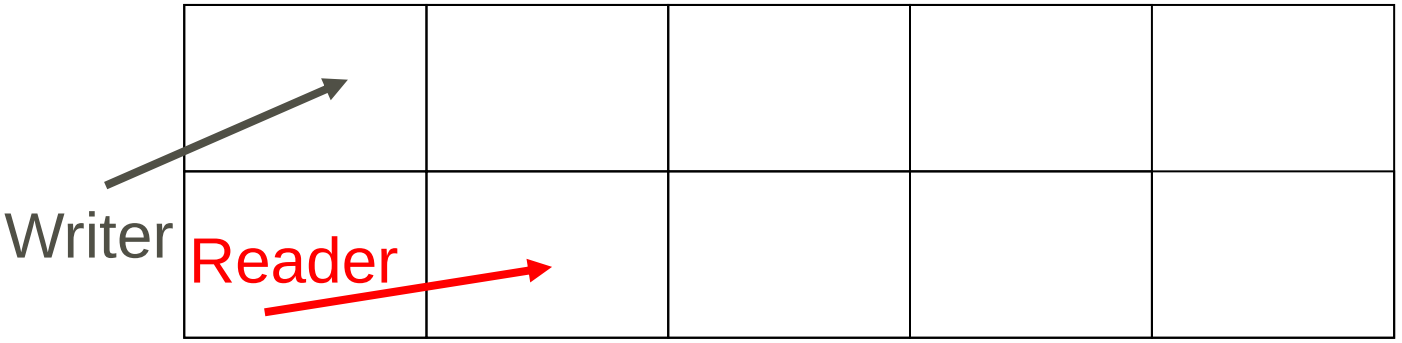
Communications Buffer

Two processes:

- Writer: Puts information out
- Reader: Gets information in

We want each process to be independent

- Asynchronous: Work at their own pace



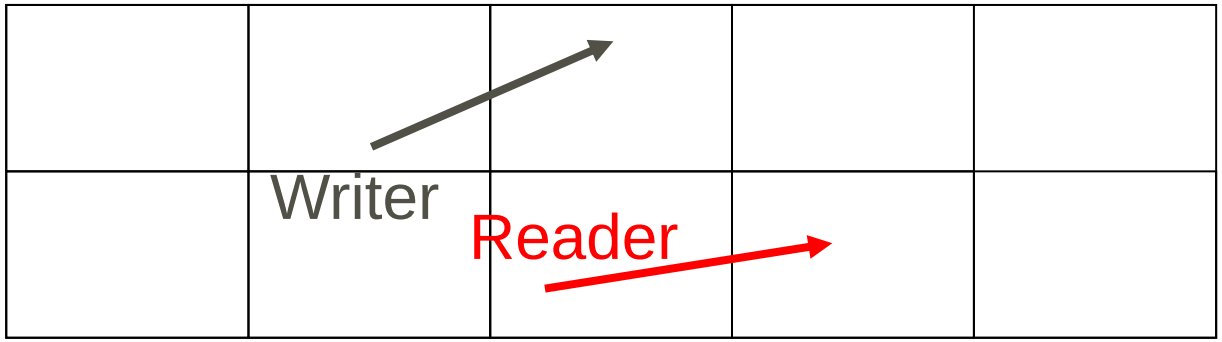
Communications Buffer

Two processes:

- Writer: Puts information out
- Reader: Gets information in

We want each process to be independent

- Asynchronous: Work at their own pace



Communications Buffer

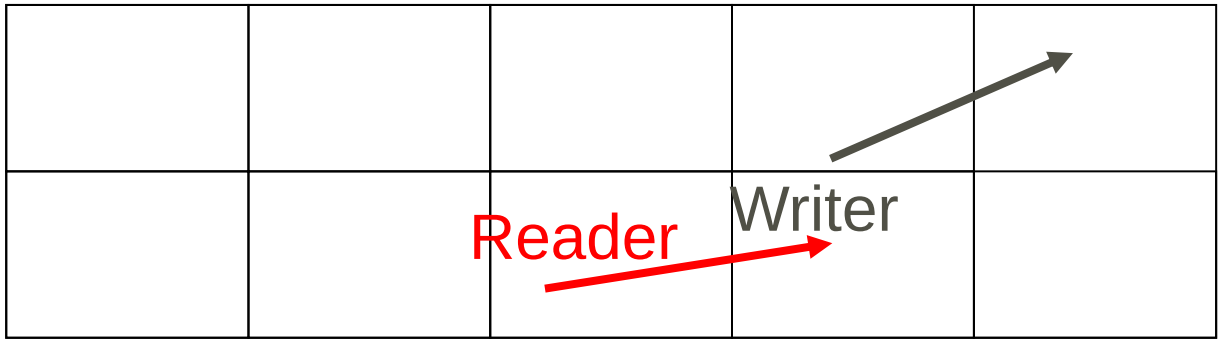
Two processes:

- Writer: Puts information out
- Reader: Gets information in

We want each process to be independent

- Asynchronous: Work at their own pace

Problem!



Communications Buffer

Two processes:

- Writer: Puts information out
- Reader: Gets information in

We want each process to be independent

- Asynchronous: Work at their own pace

Problem!

Asynchronous

Summary

Summary

The definition of the queue operations gives the ADT queue first-in, first-out (FIFO) behavior

Function:

- Enqueue(S) -> Inserts object o at the rear of the queue
- Dequeue(S) -> Removes the object from the front of the queue; an error occurs if the queue is empty
- Front(S) -> Returns the position of beginning of queue
- Rear(S)-> Returns the position of ending of queue

Types of queues:

- Simple Queue
- Circular Queue
- Priority Queue
- Double Ended Queue

Question