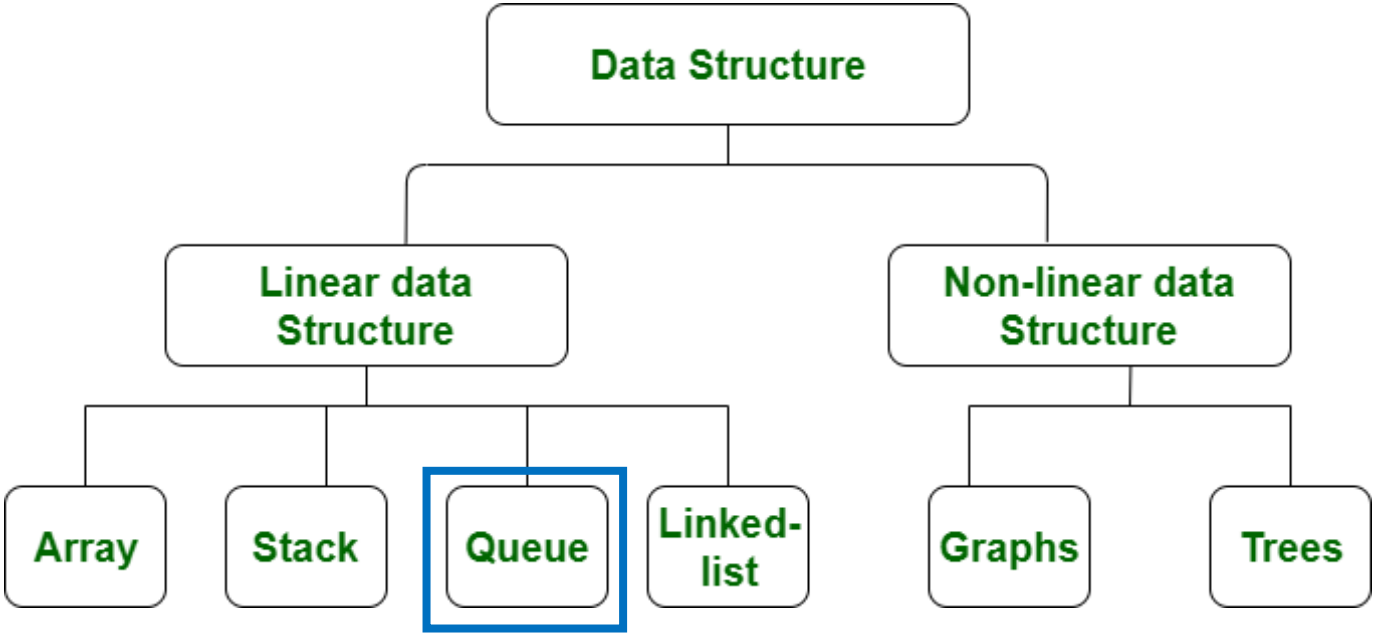
# Queue

Teaching Team of
Algorithm and Data Structure

Genap 2024/2025

# Data Structure Classification



# Definition of Queue

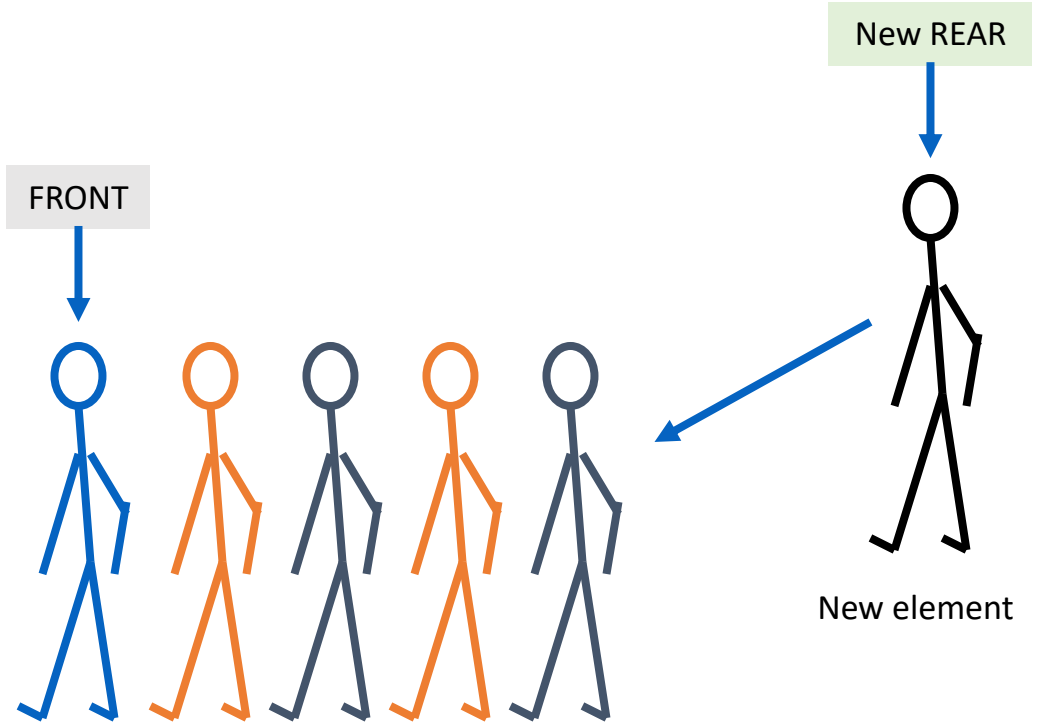
- Queue applies the principle of **FIFO (First In First Out)**
- The process of adding elements is carried out in the **rear position** and the process of taking elements is carried out in the elements in the **front position**
- Illustration of queue like people who queue to buy tickets, the first person to come will be served first

# Queue concept

- Queue has two elements:
  - The first element is called Head / Front
  - The last element is called Tail / Rear
- Adding elements is always done after the last element
- Deleting an element is always done on the first element

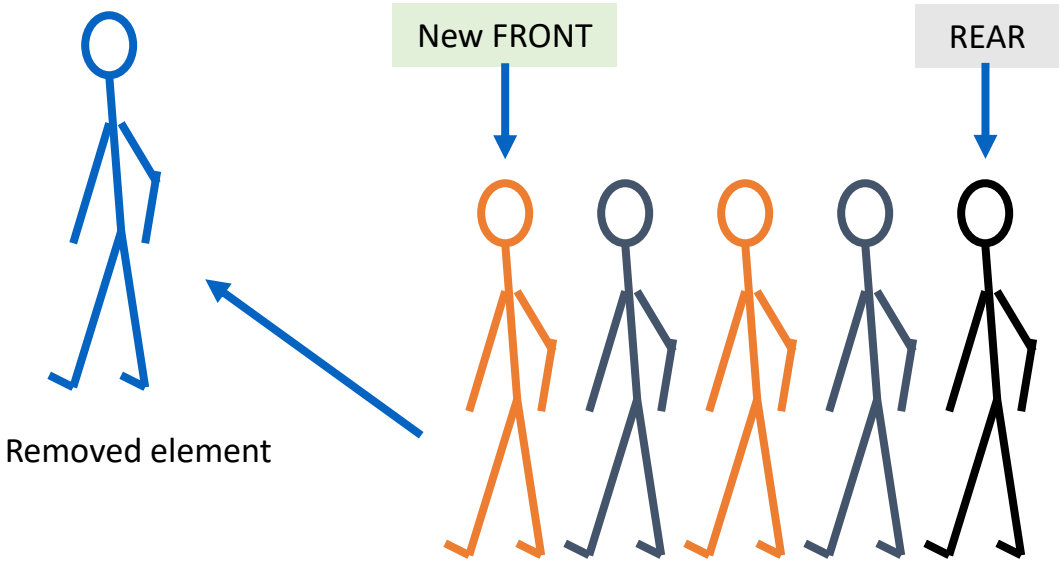
# Queue concept

- Add elements



# Queue concept

- Remove an element

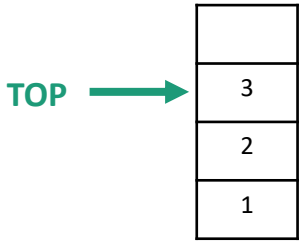


# Queue Operations

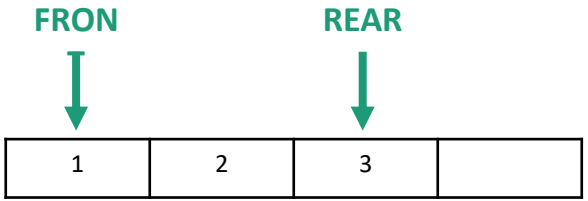
- ***Create*** : creating a new empty queue
- ***Enqueue*** : adding new data in queue
- ***Dequeue*** : removing data from
- ***Is Empty*** : checking queue whether there is no data inside
- ***Is Full*** : checking queue whether it is still capable to have a new data
- ***Peek*****:** check the data first
- ***Print*****:** display all data in the queue

# Queue Implementation

- More complex than Stack implementation
- Stack → both addition and deletion data will change one point (that is **top** position)
- Queue → addition will change **rear** position, while deletion will change **front** position




Stack




Queue

# Queue Implementation

- Using **Array**:
  - Queue length is static
  - If a queue is made with a length of 5, then the maximum queue can hold 5 data
- Using **Linked List**:
  - Queue length is dynamic
  - The amount of data that can be managed in the queue can dynamically changes based on the need

  Discussion on the Linked List will not be delivered at this meeting because it will be discussed at the next meeting

# Queue Implementation

For example there is a queue of Q with N elements → $Q_1, Q_2, \ldots, Q_N$

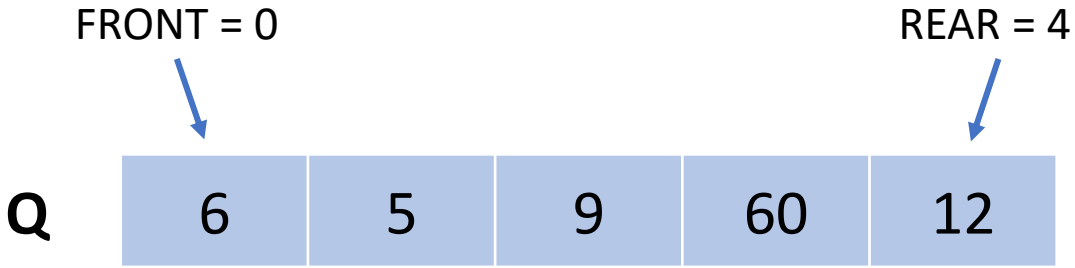
- Data in **front** of the Q → **FRONT(Q)**
- Data in the **last** position of Q → **REAR(Q)**
- The number of elements in the queue is represented by the **SIZE(Q)** symbol, which can be calculated by rear – front + 1
- For queue Q = $[Q_1, Q_2, \ldots, Q_N]$, so…
  FRONT(Q) = $Q_1$
  REAR(Q) = $Q_N$
  SIZE(Q) = N

# Queue Implementation with Arrays

- **Q :** attribute/variabel that will save queue data
- **FRONT** : attribute/variabel that will save **index of array**, where **FRONT** data takes place
- **REAR** : attribute/variabel that will save **index of array**, where **REAR** data takes place
- **SIZE** : attribute/variabel that will save the number of data that are currently in the queue
- **MAX** : attribute/variabel that will save the maximum number of data could be managed by queue

# Queue Implementation with Array

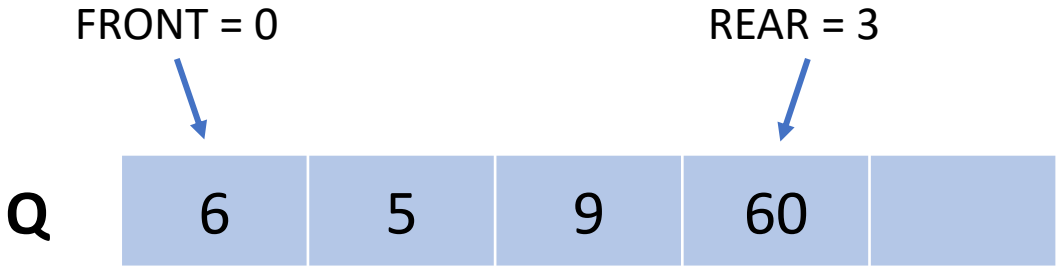
- Illustration when queue is full




- SIZE = 5
- MAX = 5
- The queue is fully loaded and cannot receive queue data anymore

# Queue Implementation with Array

- Illustration when the queue is not full




- SIZE = 4
- MAX = 5
- Queue is not full so it can still receive queue data again

# Queue Declaration

- Declaring new queue to manage the data
- Steps:
  - Class declaration
  - Front and rear declaration
  - Size and max declaration
  - Array declaration

```
public class Queue {
    int front;
    int rear;
    int size;
    int max;
    int[] Q;
}
```

# Create operation

- To initialize a queue, the variable that needs to be initialized are
  - **size** 0 because the array is still empty
  - **front** and **rear** = -1 because there is no data, points to index -1

```
public void Create() {
    Q = new int[max];
    size = 0;
    front = rear = -1;
}
```

# IsFull Operation

- To check whether the queue is in **full** condition by checking the **size**
- If the **size** is the **same as max**, then **full**
- If the **size** is still **smaller** than the **max**, then it's **not full**

```
public boolean IsFull() {
    if (size == max) {
        return true;
    } else {
        return false;
    }
}
```

# IsEmpty operation

- To check whether the queue is empty by checking the **size**
- If the **size** is still equal to 0, then the stack is still empty

```
public boolean IsEmpty() {
    if (size == 0) {
        return true;
    } else {
        return false;
    }
}
```

# Peek Operation

- To access the element in the front position (not always at index [0])

```
public void peek() {
    if (!IsEmpty()) {
        System.out.println("Elemen terdepan: " + Q[front]);
    } else {
        System.out.println("Antrian masih kosong");
    }

}
```

# Print Operation

- To display all data in the queue
- The process is done by looping all contents of the array starting from index **front** to the index **rear**. Looping is not always done from index [0] because the front is not always at index [0]

```
public void print() {
    if (IsEmpty()) {
        System.out.println("Antrian masih kosong");
    } else {
        int i = front;
        while (i != rear) {
            System.out.print(Q[i] + " ");
            i = (i + 1) % max;
        }
        System.out.println(Q[i] + " ");
        System.out.println("Jumlah antrian = " + size);
    }
```

# Enqueue Operation

- To add new data to the queue
- In the enqueue process, new data will be added at the final position in the queue
- There are 3 possible conditions that occur during Enqueue:
  - When the queue is empty
  - When the **rear** of the queue is not in the last index of the array
  - When the **rear** of the queue is in the last index of the array

# Enqueue Operation - Condition 1

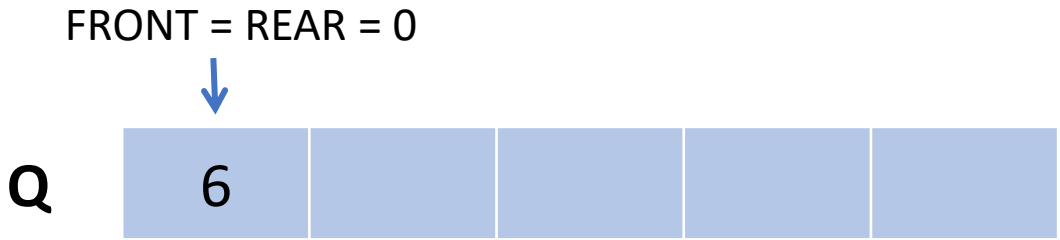
## 1. When the queue is empty

FRONT = REAR = -1


- SIZE = 0
- MAX = 5

# Enqueue Operation - Condition 1

- When data is added, new data is entered into the queue at index 0.
- The data becomes data in the position of FRONT and REAR




- SIZE = 1
- MAX = 5

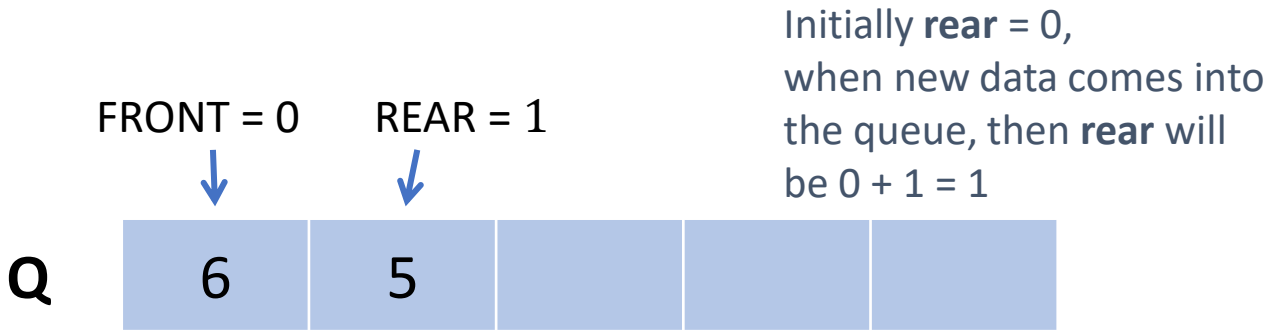
# Enqueue Operation - Condition 2

## 2. When the rear of the queue is not in the last index of the array


- SIZE = 1
- MAX = 5

# Enqueue Operation - Condition 2

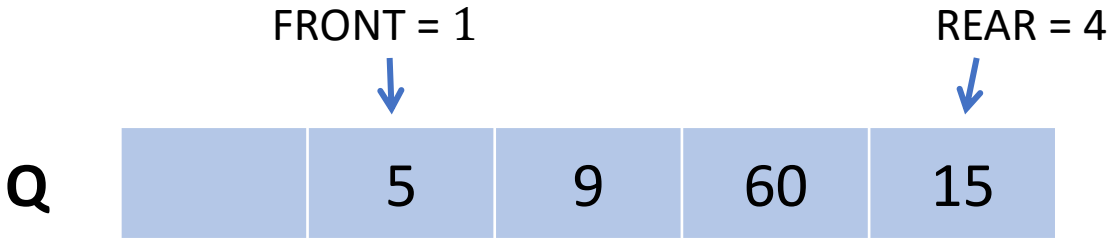
- When new data is entered, the data will be located right after the current **rear** position, which is in the index REAR +1




- SIZE = 2
- MAX = 5

# Enqueue Operation - Condition 3

## 3. When the rear of the queue is in the last index of the array

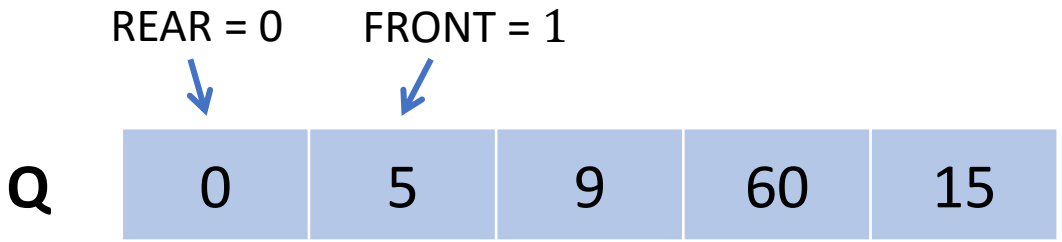



- SIZE = 4
- MAX = 5

Note that the front is not always at index [0], it could be index [1] because previously there was already a data deleted

# Enqueue Operation - Condition 3

- When the new data is entered, the data will be located at index [0], which is in the index REAR = 0




- SIZE = 5
- MAX = 5

# Enqueue Operations Algorithm

- Ensure that the queue is not in full condition. **If the queue is full**, then the **data cannot be entered** into the queue.
- **If it is not full**, then we can continue to perform data addition.
  - Check whether the **queue is empty**. If the queue is empty, it means that the data will be entered into index [0] and it will become **front** as well as **rear** data. Which is FRONT = REAR = 0
  - If the **queue isn't empty**, then:
    - Checks whether the **REAR** is at the last index of the array. If true, then the next **REAR** position will be at index [0]
    - If the **REAR** is not at the last array index, then the next **REAR** position will be **REAR + 1**
  - Enter data into the queue in the **REAR** index
- SIZE increased by 1

# Implementation of Enqueue

```
public void Enqueue(int data) {
    if (IsFull()) {
        System.out.println("Queue sudah penuh");
    } else {
        if (IsEmpty()) {
            front = rear = 0;
        } else {
            if (rear == max - 1) {
                rear = 0;
            } else {
                rear++;
            }
        }
        Q[rear] = data;
        size++;
    }
}
```

Enqueue cond. 1

Enqueue cond. 3

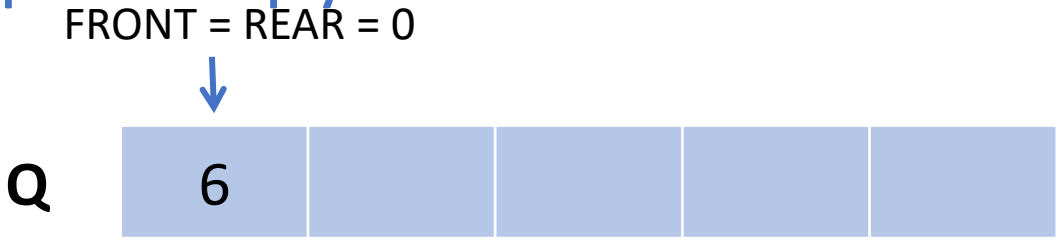
Enqueue cond. 2

# Operation Dequeue

- To retrieve data from the queue
- In the dequeue process, the data that will be retrieved is the data that is in the **front** position of the queue
- There are 3 possible conditions that occur when Dequeue:
  - When the queue is empty after the data is retrieved
  - When the **front** data is not in the last index of the array
  - When the **front** data is in the last index of the array

# Dequeue Operation - Condition 1

**1. When the queue is empty after the data is retrieved**




- SIZE = 1
- MAX = 5

# Dequeue Operation - Condition 1

- After the retrieval we will get 6, and the FRONT and REAR positions are set to -1

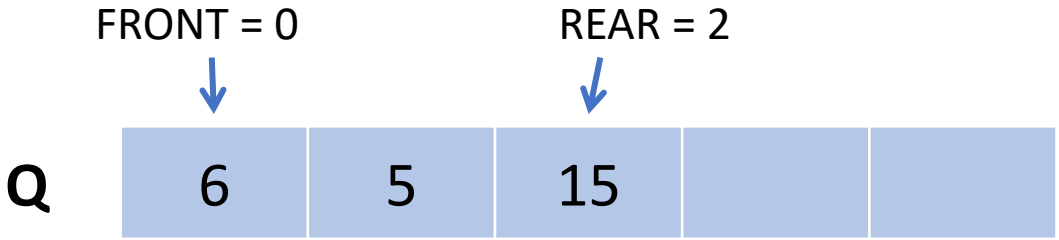
FRONT = REAR = -1

Q

- SIZE = 0
- MAX = 5

# Dequeue Operation - Condition 2

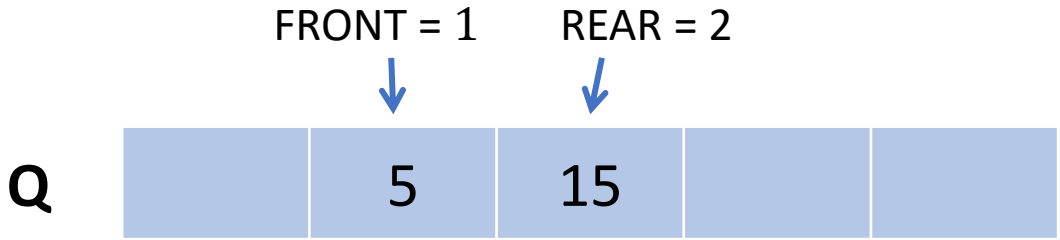
## 2. When the front data is not in the last index of the array




- SIZE = 3
- MAX = 5

# Dequeue Operation - Condition 2

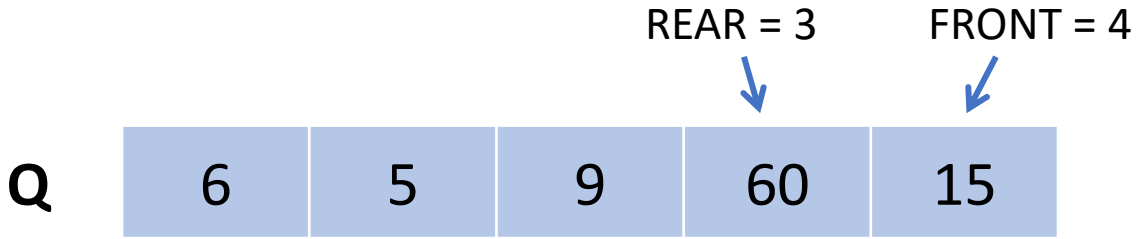
- The 6 is retrieved, and then the FRONT position will be increased by 1 from the previous position




- SIZE = 2
- MAX = 5

# Dequeue Operation - Condition 3

## 3. When the front data is in the last index of the array

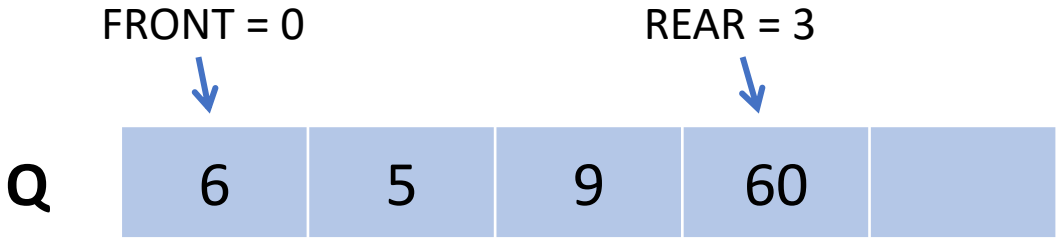



- SIZE = 5
- MAX = 5

Note: that the front index can be larger than the rear because in full condition there is a deletion of data until the front is in index [4], then the addition of data is carried out so that it shifts the rear index

# Dequeue Operation - Condition 3

- We will get 15 from the retrieval, and the position of FRONT will be shifted to index [0]




- SIZE = 4
- MAX = 5

# Dequeue Operations Algorithm

- Ensure that the queue is not empty. **If the queue is empty**, no data can be retrieved
- **If it is not empty**, then the process for retrieving data from the queue can be performed.
  - Take the data that is in the **FRONT** index, where the data will be returned from this process
  - **SIZE** decreases by 1
  - Next, change the position of **FRONT**:
    - Check whether after retrieving the data, the **queue is empty** (SIZE = 0). If true, then the position **FRONT** = **REAR** = -1
    - If after the data is retrieved and the **queue is not empty**, then:
      - Checks whether the current position of **FRONT** is **in the last index of the array**. If true, then the next **FRONT** is located at index 0
      - If the **FRONT** position is **not in the last index of the array**, then the next **FRONT** position is the previous **FRONT plus 1**

# Implementation of Dequeue

```
public int Dequeue() {
    int data = 0;
    if (IsEmpty()) {
        System.out.println("Queue masih kosong");
    } else {
        data = Q[front];
        size--;
        if (IsEmpty()) {
            front = rear = -1;
        } else {
            if (front == max - 1) {
                front = 0;
            } else {
                front++;
            }
        }
    }
    return data;
}
```

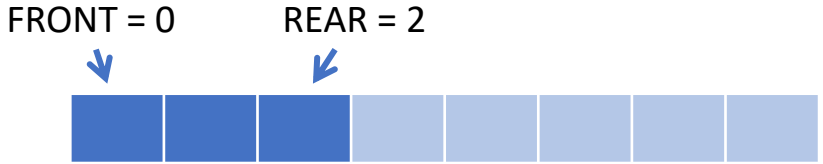
Dequeue cond. 1

Dequeue cond. 3

Dequeue cond. 2

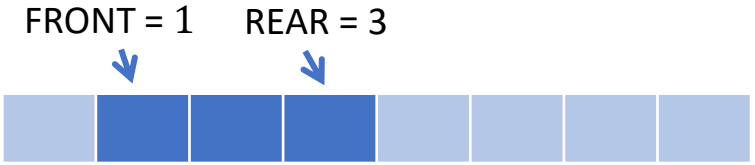
# Front and Rear Changes

- Front index increases by 1 every time Dequeue occurs
- The rear index increases by 1 each time Enqueue occurs
- Current condition:



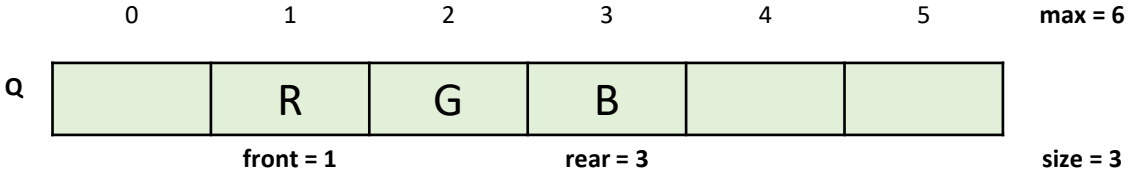

- After Enqueue:


- After Dequeue:



# Assignments

1. The following queue has 6 capacities to mange data:




   Draw the queue illustration for the following operations:
   - Add A
   - Delete R and G
   - Add X, Y, and Z
   - Delete B and A
2. Create the flowchart for Enqueue and Dequeue!