Data Structures
COMP242

Ala' Hasheesh
ahashesh@birzeit.edu

Queues

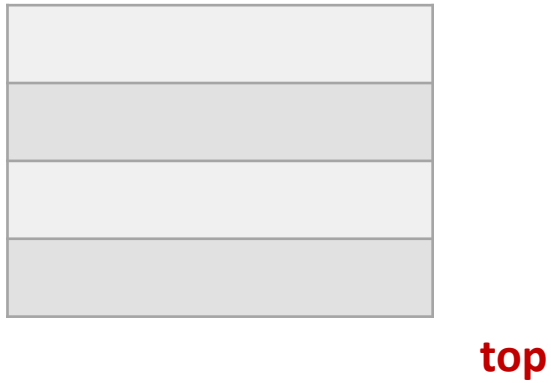
Stack

A **stack** is a data structure in which elements are added and removed from one end!

It can allow us to insert elements and retrieve them in opposite order!

Operations

1. Push: Insert/Add a new element to **top** the stack
2. Pop: Remove/Delete **top** element in the stack
3. Peek: Examine element at the **top** of the stack



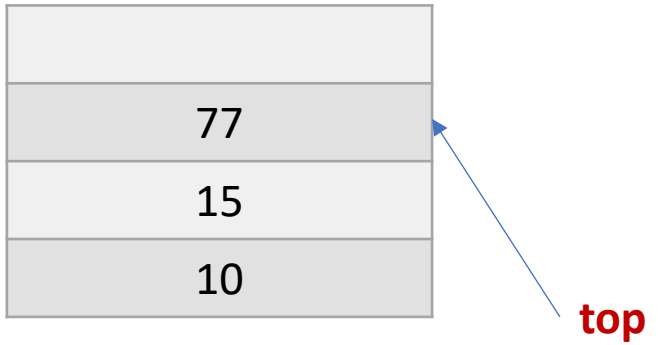
Stack

Stacks are called (**LIFO**): Last In First Out

Last element that gets pushed is the first element that gets popped!

They are also called (**FILO**): First In Last Out

First element that gets pushed is the last element that gets popped!



Queue

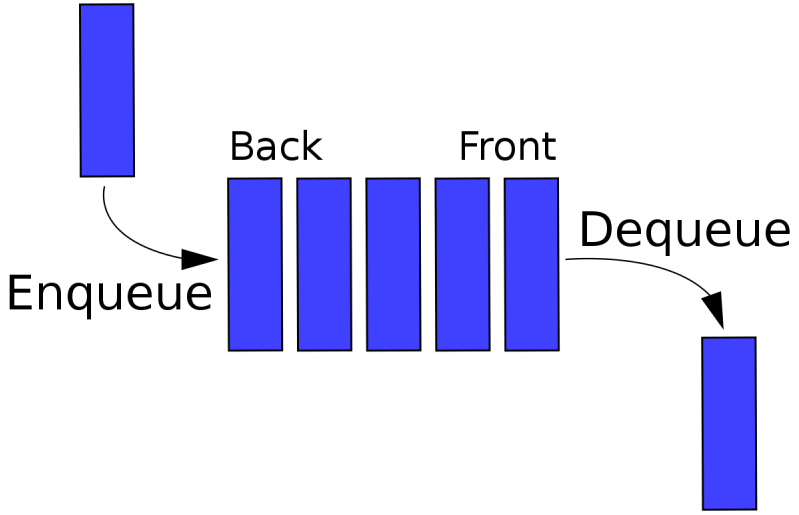
Queues are called (**FIFO**): First In First Out

First element that is enqueued/inserted is the first element that gets dequeued/removed!

Queue

Queues are called (**FIFO**): First In First Out

First element that is enqueued/inserted is the first element that gets dequeued/removed!

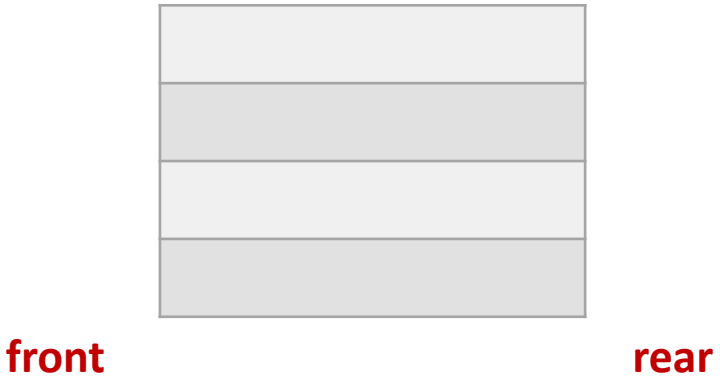


Queue

A **queue** is a data structure in which elements are added from one end (e.g. Front)
And removed from the other end (e.g. Rear)

Operations

1. Enqueue/insert: Insert/Add a new element to **rear** element in the queue
2. Dequeue/remove: Remove/Delete **front** element in the queue
3. peek: Examine element at the **front** of the queue
4. Others: size, isEmpty, …



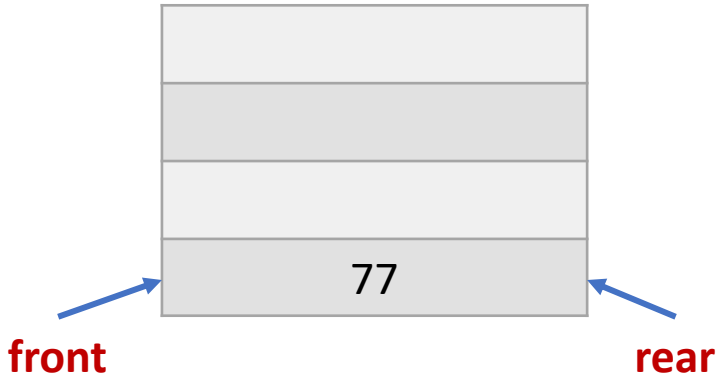
Queue

A **queue** is a data structure in which elements are added from one end (e.g. Front)
And removed from the other end (e.g. Rear)

Operations

1. Enqueue/insert: Insert/Add a new element to **rear** element in the queue
2. Dequeue/remove: Remove/Delete **front** element in the queue
3. peek: Examine element at the **front** of the queue
4. Others: size, isEmpty, …

queue.enqueue(77);



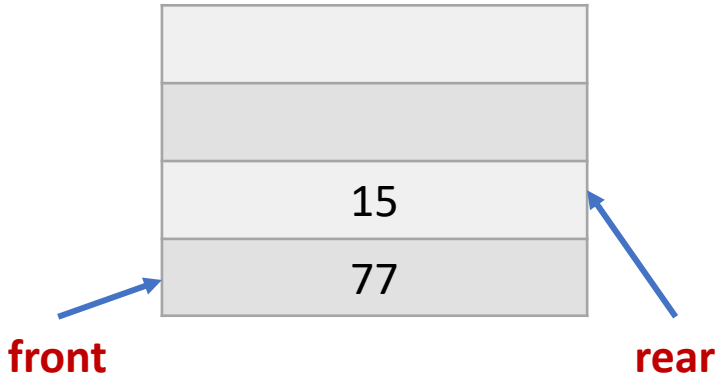
Queue

A **queue** is a data structure in which elements are added from one end (e.g. Front)
And removed from the other end (e.g. Rear)

Operations

1. Enqueue/insert: Insert/Add a new element to **rear** element in the queue
2. Dequeue/remove: Remove/Delete **front** element in the queue
3. peek: Examine element at the **front** of the queue
4. Others: size, isEmpty, …

queue.enqueue(15);



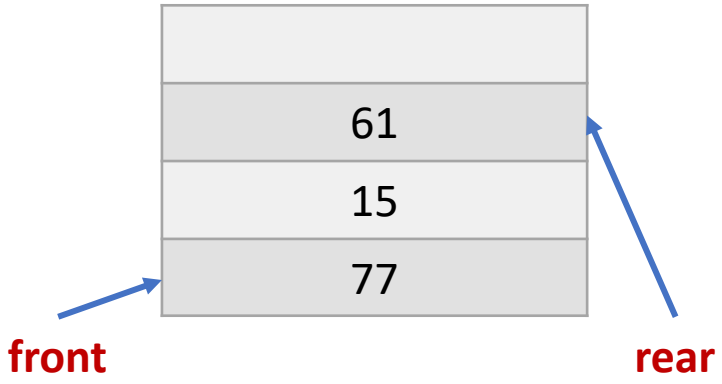
Queue

A **queue** is a data structure in which elements are added from one end (e.g. Front)
And removed from the other end (e.g. Rear)

Operations

1. Enqueue/insert: Insert/Add a new element to **rear** element in the queue
2. Dequeue/remove: Remove/Delete **front** element in the queue
3. peek: Examine element at the **front** of the queue
4. Others: size, isEmpty, …

queue.enqueue(61);



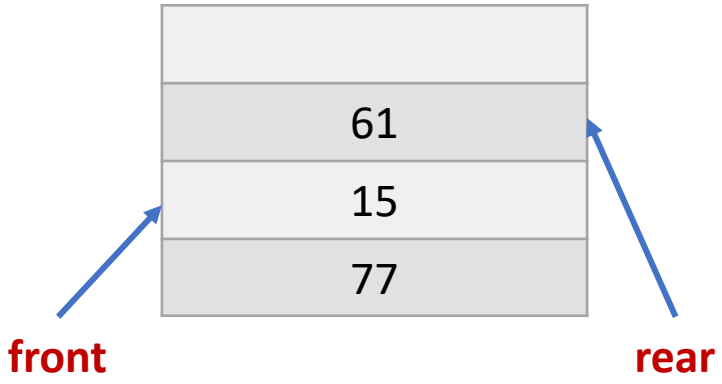
Queue

A **queue** is a data structure in which elements are added from one end (e.g. Front)
And removed from the other end (e.g. Rear)

Operations

1. Enqueue/insert: Insert/Add a new element to **rear** element in the queue
2. Dequeue/remove: Remove/Delete **front** element in the queue
3. peek: Examine element at the **front** of the queue
4. Others: size, isEmpty, …

queue.dequeue();



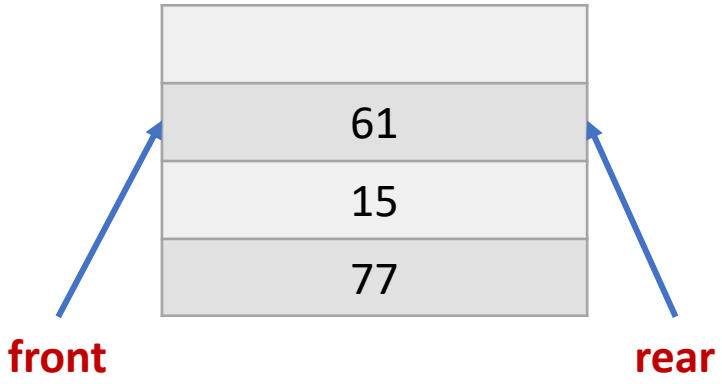
Queue

A **queue** is a data structure in which elements are added from one end (e.g. Front)
And removed from the other end (e.g. Rear)

Operations

1. Enqueue/insert: Insert/Add a new element to **rear** element in the queue
2. Dequeue/remove: Remove/Delete **front** element in the queue
3. peek: Examine element at the **front** of the queue
4. Others: size, isEmpty, …

queue.dequeue();



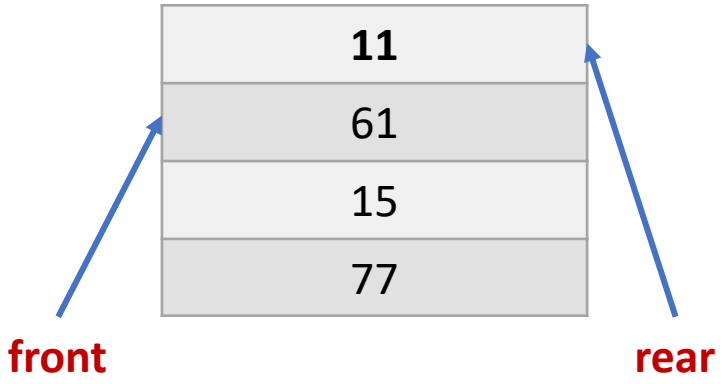
Queue

A **queue** is a data structure in which elements are added from one end (e.g. Front)
And removed from the other end (e.g. Rear)

Operations

1. Enqueue/insert: Insert/Add a new element to **rear** element in the queue
2. Dequeue/remove: Remove/Delete **front** element in the queue
3. peek: Examine element at the **front** of the queue
4. Others: size, isEmpty, …

queue.enqueue(11);



Queue

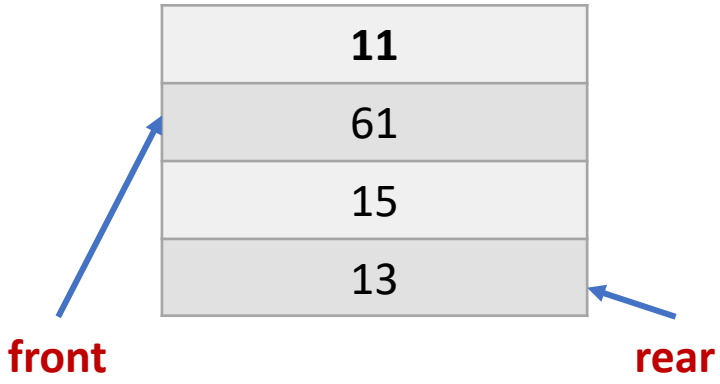
A **queue** is a data structure in which elements are added from one end (e.g. Front)
And removed from the other end (e.g. Rear)

Operations

1. Enqueue/insert: Insert/Add a new element to **rear** element in the queue
2. Dequeue/remove: Remove/Delete **front** element in the queue
3. peek: Examine element at the **front** of the queue
4. Others: size, isEmpty, …

Imagine the array is circular and we reach the end we reset back to 0.
(Last element is connected with the first one)

queue.enqueue(13);



Queue (Applications)

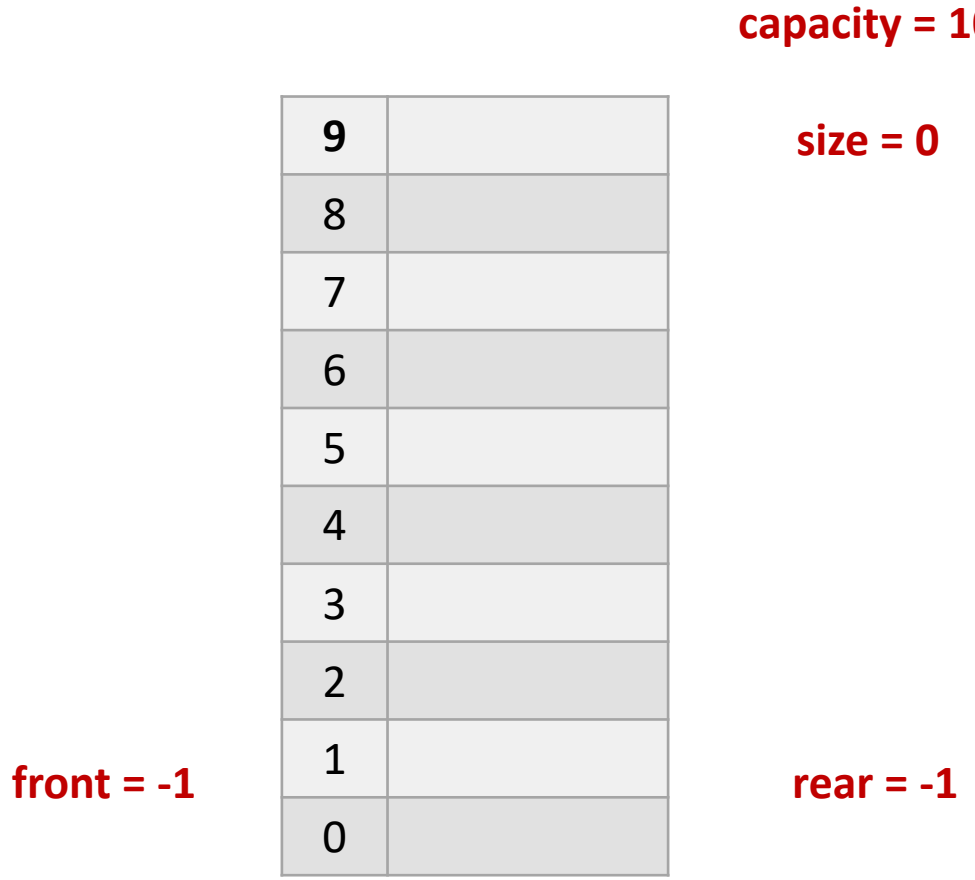
1. Real world examples:
    1. People waiting inside a bank (without priority for now).
    2. People waiting in a doctor office/barber.
    3. Cars in a single small road.

2. Operating System:
    1. Tasks to print.
    2. Simple process scheduling/execution.

3. Programming:
    1. Create a system to model line of customers/orders.
    2. Breadth First Search

Queue (Array Implementation)

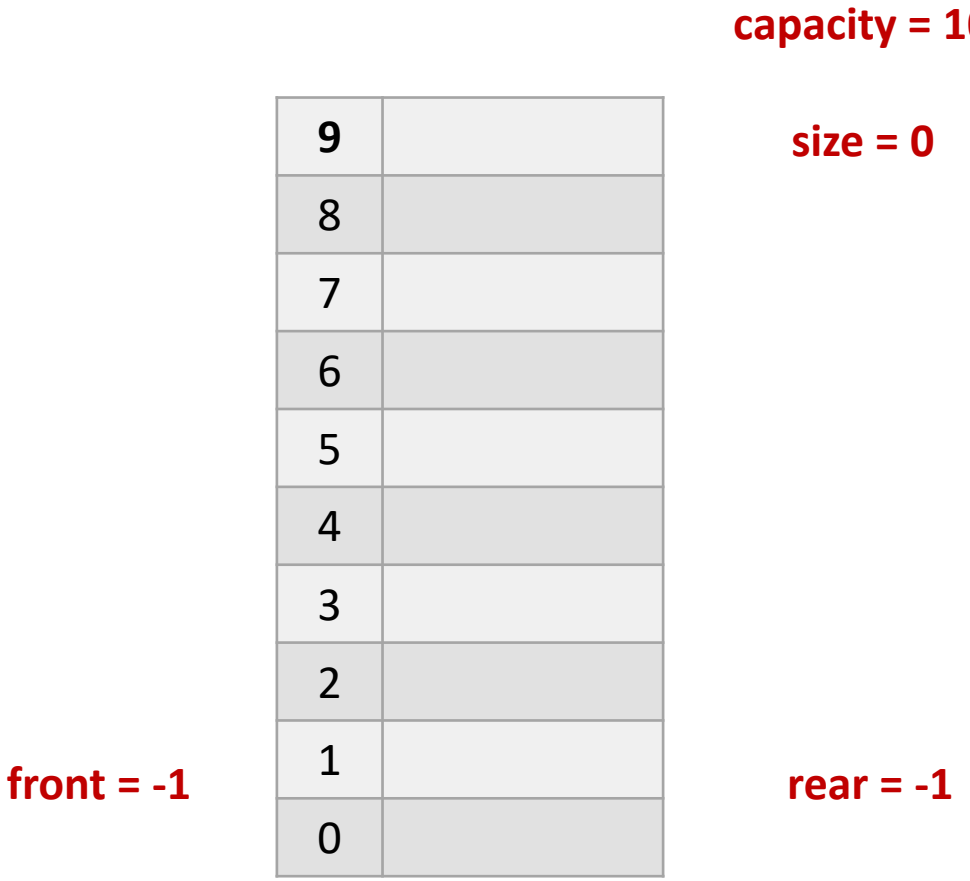
```
public class Queue<T> {
    private int capacity;
    private int size;
    private int front;
    private int rear;
    T[] array;

    public Queue() {
        this(10);
    }

    public Queue(int capacity) {
        this.capacity = capacity;
        this.size = 0;
        this.front = -1;
        this.rear = -1;
        array = (T[]) new Object[capacity];
    }
}
```




```
Queue<Integer> queue = new Queue<>();
```

Queue (Array Implementation)

```
public boolean isEmpty() {
    return (front == -1 && rear == -1) || size == 0;
}
```

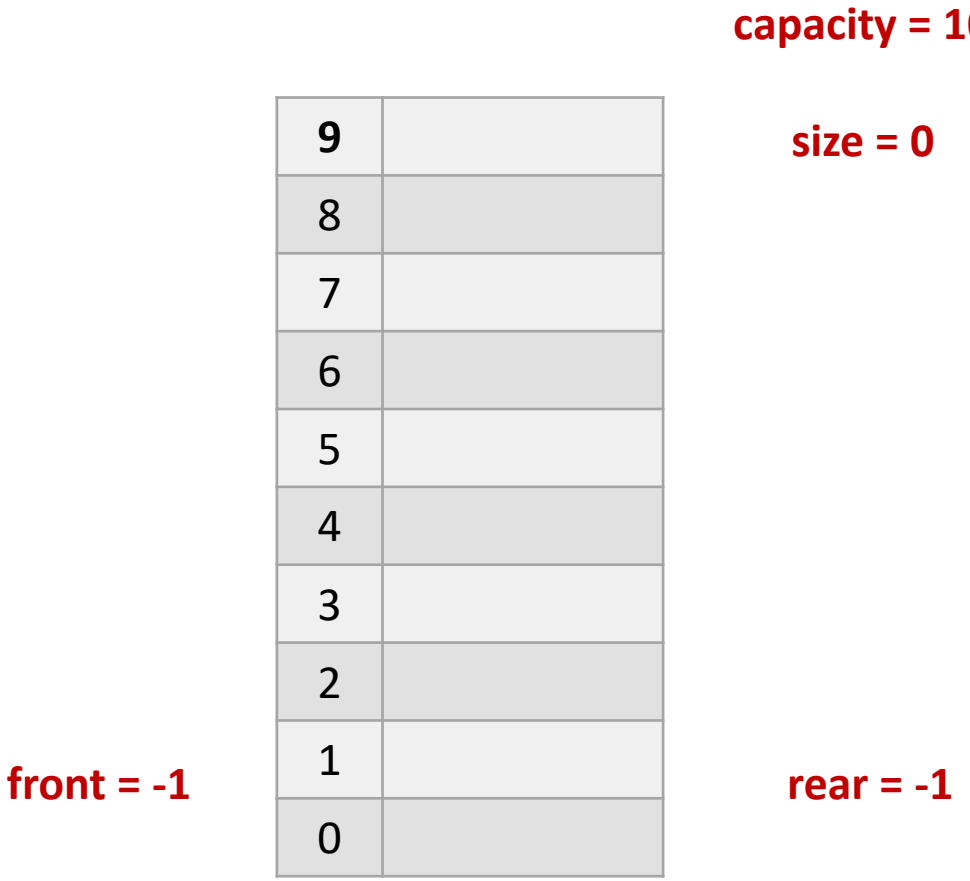



```
Queue<Integer> queue = new Queue<>();
```

Queue (Array Implementation)

```
public boolean isEmpty() {
    return (front == -1 && rear == -1) || size == 0;
}

public boolean isFull() {
    return rear + 1 == front || size == capacity;
}
```

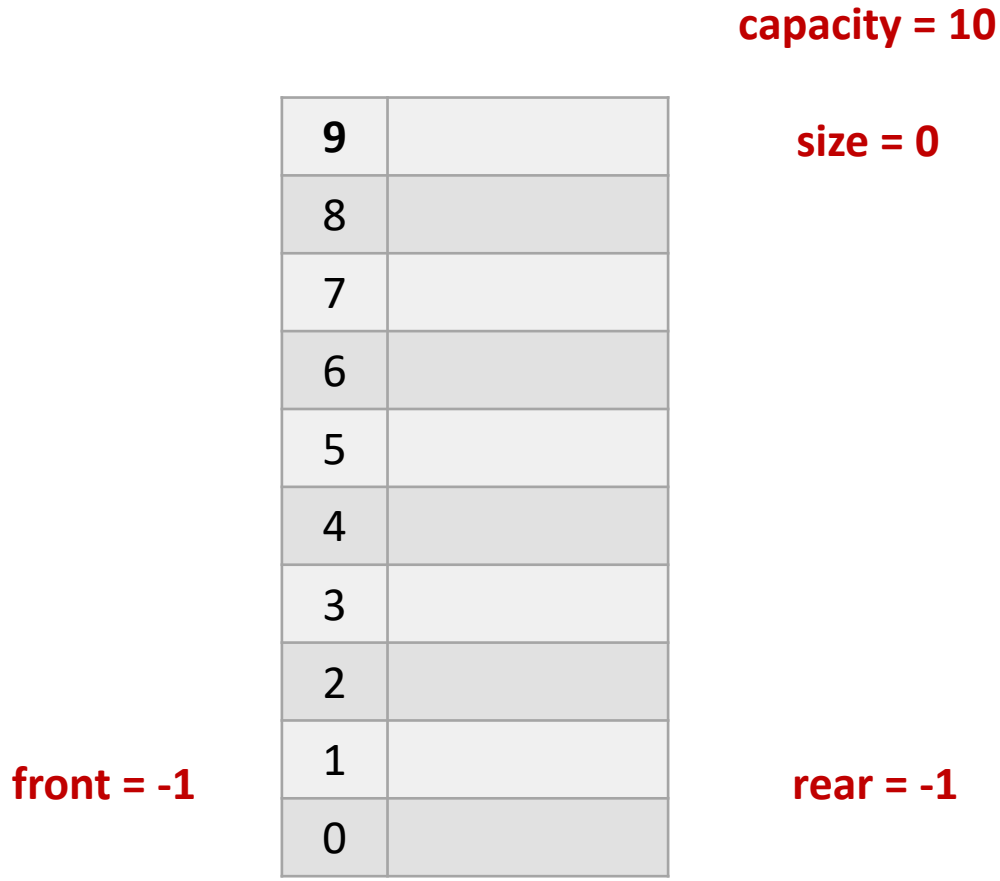
How do we make array circular?



Queue<Integer> queue = new Queue<>();

Queue (Array Implementation)

```
public boolean isEmpty() {
    return (front == -1 && rear == -1) || size == 0;
}

public boolean isFull() {
    return (rear + 1) % capacity == front || size == capacity;
}
```



Queue<Integer> queue = new Queue<>();

Queue (Array Implementation)

```
public boolean isEmpty() {
    return (front == -1 && rear == -1) || size == 0;
}

public boolean isFull() {
    return (rear + 1) % capacity == front || size == capacity;
}

public void enqueue(T element) {
    rear = (rear + 1) % capacity;
    array[rear] = element;
    size++;
}
```

What happens if array is full?

capacity = 10

size = 0

9	
8	
7	
6	
5	
4	
3	
2	
1	
0	

front = -1

rear = -1

```
Queue<Integer> queue = new Queue<>();
```

Queue (Array Implementation)

```
public boolean isEmpty() {
    return (front == -1 && rear == -1) || size == 0;
}

public boolean isFull() {
    return (rear + 1) % capacity == front || size == capacity;
}

public void enqueue(T element) {
    rear = (rear + 1) % capacity;
    array[rear] = element;
    size++;
}
```

What happens if array is full?

capacity = 10

size = 0

9	
8	
7	
6	
5	
4	
3	
2	
1	
0	

front = -1

rear = -1

```
Queue<Integer> queue = new Queue<>();
```

Queue (Array Implementation)

```
public boolean isEmpty() {
    return (front == -1 && rear == -1) || size == 0;
}

public boolean isFull() {
    return (rear + 1) % capacity == front || size == capacity;
}

public void enqueue(T element) {
    if (isFull()) {
        throw new OutOfMemoryError("Queue is Full");
    }

    rear = (rear + 1) % capacity;
    array[rear] = element;
    size++;
}
```

capacity = 10

9	
8	
7	
6	
5	
4	
3	
2	
1	
0	

size = 0

front = -1

rear = -1

```
Queue<Integer> queue = new Queue<>();
```

What happens if array is empty? front = -1 and rear = -1

Queue (Array Implementation)

```
public boolean isEmpty() {
    return (front == -1 && rear == -1) || size == 0;
}

public boolean isFull() {
    return (rear + 1) % capacity == front || size == capacity;
}

public void enqueue(T element) {
    if (isFull()) {
        throw new OutOfMemoryError("Queue is Full");
    }

    rear = (rear + 1) % capacity;
    array[rear] = element;
    size++;
}
```

capacity = 10

9	
8	
7	
6	
5	
4	
3	
2	
1	
0	

size = 0

front = -1

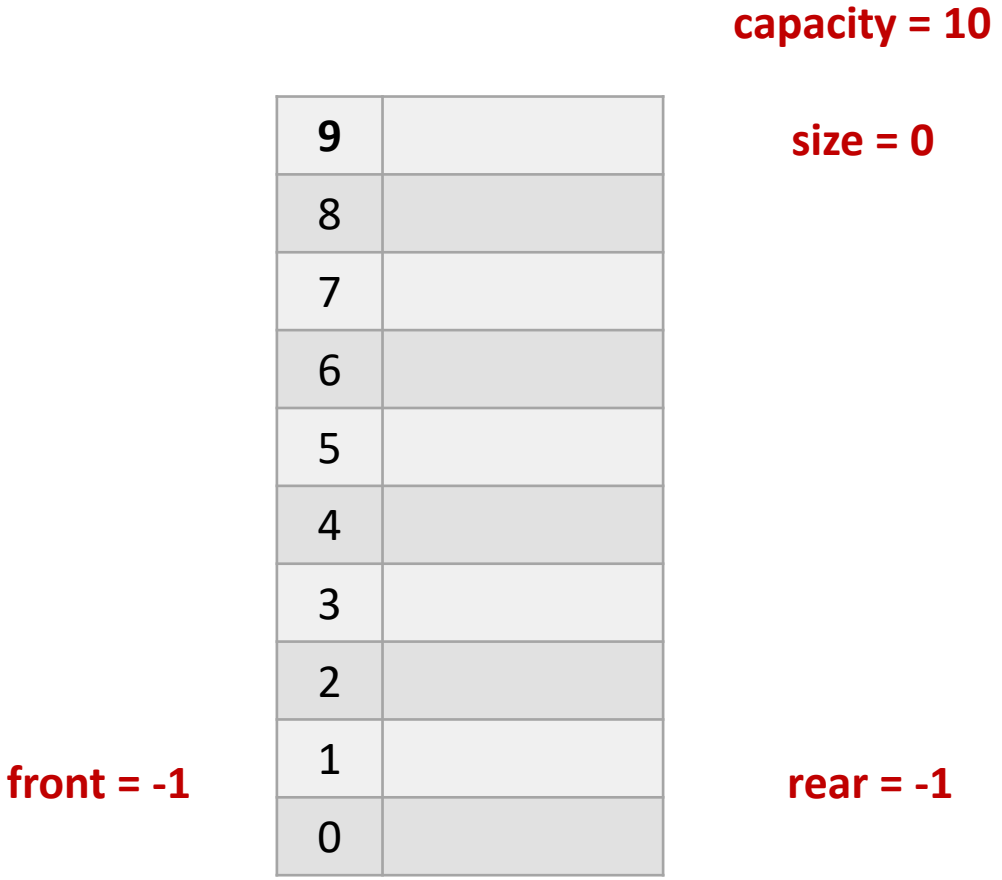
rear = -1

Queue<Integer> queue = new Queue<>();

What happens if array is front = -1 and rear = -1

Queue (Array Implementation)

```
public boolean isEmpty() {
   return (front == -1 && rear == -1) || size == 0;
}

public boolean isFull() {
   return (rear + 1) % capacity == front || size == capacity;
}

public void enqueue(T element) {
   if (isFull()) {
      throw new OutOfMemoryError("Queue is Full");
   }

   if (front == -1) {
      front++;
   }

   rear = (rear + 1) % capacity;
   array[rear] = element;
   size++;
}
```




```
Queue<Integer> queue = new Queue<>();
```

Queue (Array Implementation)

```
public boolean isEmpty() {
    return (front == -1 && rear == -1) || size == 0;
}

public boolean isFull() {
    return (rear + 1) % capacity == front || size == capacity;
}

public void enqueue(T element) {
    if (isFull()) {
        throw new OutOfMemoryError("Queue is Full");
    }

    if (front == -1) {
        front++;
    }

    rear = (rear + 1) % capacity;
    array[rear] = element;
    size++;
}
```

capacity = 10

size = 1

9	
8	
7	
6	
5	
4	
3	
2	
1	
0	10

front = 0

rear = 0

```
queue.enqueue(10);
```

```
Queue<Integer> queue = new Queue<>();
```

Queue (Array Implementation)

```
public boolean isEmpty() {
   return (front == -1 && rear == -1) || size == 0;
}

public boolean isFull() {
   return (rear + 1) % capacity == front || size == capacity;
}

public void enqueue(T element) {
   if (isFull()) {
      throw new OutOfMemoryError("Queue is Full");
   }

   if (front == -1) {
      front++;
   }

   rear = (rear + 1) % capacity;
   array[rear] = element;
   size++;
}
```

capacity = 10

size = 2

9	
8	
7	
6	
5	
4	
3	
2	
1	13
0	10

front = 0

rear = 1

```
queue.enqueue(13);
```

```
Queue<Integer> queue = new Queue<>();
```

Queue (Array Implementation)

```
public boolean isEmpty() {
    return (front == -1 && rear == -1) || size == 0;
}

public boolean isFull() {
    return (rear + 1) % capacity == front || size == capacity;
}

public void enqueue(T element) {
    if (isFull()) {
        throw new OutOfMemoryError("Queue is Full");
    }

    if (front == -1) {
        front++;
    }

    rear = (rear + 1) % capacity;
    array[rear] = element;
    size++;
}
```

capacity = 10

size = 3

9	
8	
7	
6	
5	
4	
3	
2	7
1	13
0	10

front = 0

rear = 2

```
queue.enqueue(7);
```

```
Queue<Integer> queue = new Queue<>();
```

Queue (Array Implementation)

```
public T dequeue() {
    if (isEmpty()) {
        return null;
    }

    T element = array[front];
    if (front == rear) {
        front = rear = -1;
    } else {
        front = (front + 1) % capacity;
    }

    size--;
    return element;
}
```

capacity = 10

size = 3

9	
8	
7	
6	
5	
4	
3	
2	7
1	13
0	10

front = 0

rear = 2

```
Queue<Integer> queue = new Queue<>();
```

Queue (Array Implementation)

```
public T dequeue() {
    if (isEmpty()) {
        return null;
    }

    T element = array[front];
    if (front == rear) {
        front = rear = -1;
    } else {
        front = (front + 1) % capacity;
    }

    size--;
    return element;
}
```

queue.dequeue();

capacity = 10

size = 2

9	
8	
7	
6	
5	
4	
3	
2	7
1	13
0	~~10~~

front = 1

rear = 2

Queue<Integer> queue = new Queue<>();

Queue (Array Implementation)

```
public T dequeue() {
    if (isEmpty()) {
        return null;
    }

    T element = array[front];
    if (front == rear) {
        front = rear = -1;
    } else {
        front = (front + 1) % capacity;
    }

    size--;
    return element;
}
```

queue.dequeue();

capacity = 10

size = 1

9	
8	
7	
6	
5	
4	
3	
2	7
1	~~13~~
0	~~10~~

front = 2

rear = 2

Queue<Integer> queue = new Queue<>();

Queue (Array Implementation)

```
public T dequeue() {
    if (isEmpty()) {
        return null;
    }

    T element = array[front];
    if (front == rear) {
        front = rear = -1;
    } else {
        front = (front + 1) % capacity;
    }

    size--;
    return element;
}
```

queue.dequeue();

capacity = 10

size = 0

9	
8	
7	
6	
5	
4	
3	
2	~~7~~
1	~~13~~
0	~~10~~

front = -1

rear = -1

Queue<Integer> queue = new Queue<>();

Queue (Array Implementation)

```
public T front() {
   if (isEmpty()) {
      return null;
   }

   return array[front];
}
```

Get element at the front of the queue!

capacity = 10

size = 0

9	
8	
7	
6	
5	
4	
3	
2	~~7~~
1	~~13~~
0	~~10~~

front = -1

rear = -1

```
Queue<Integer> queue = new Queue<>();
```

Queue (Array Implementation)

```
public T front() {
    if (isEmpty()) {
        return null;
    }

    return array[front];
}

public T rear() {
    if (isEmpty()) {
        return null;
    }

    return array[rear];
}
```

Get element at the back of the queue!

capacity = 10

size = 0

9	
8	
7	
6	
5	
4	
3	
2	~~7~~
1	~~13~~
0	~~10~~

front = -1

rear = -1

```
Queue<Integer> queue = new Queue<>();
```

Queue (Array Implementation)

```
public void print() {
   // This is an exercise
}

public void clear() {
   // This is an exercise
}
```

This is an exercise

1. **You need to know the size**
2. **You could use (front + i) % capacity**

capacity = 10

size = 0

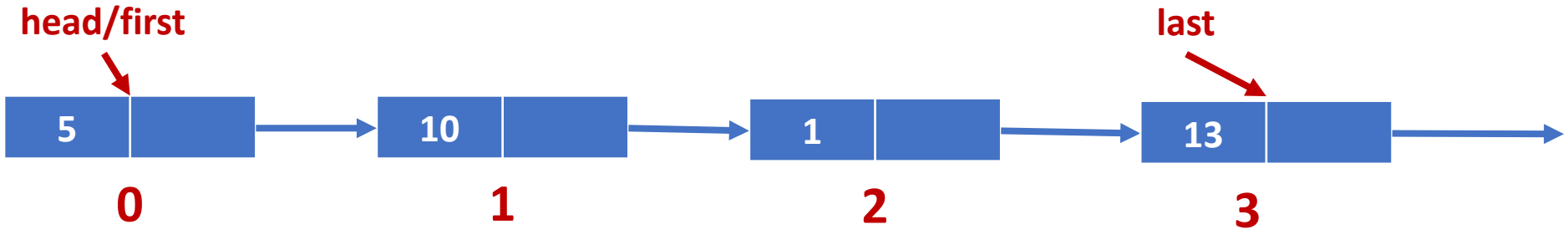
9	
8	
7	
6	
5	
4	
3	
2	~~7~~
1	~~13~~
0	~~10~~

front = -1

rear = -1

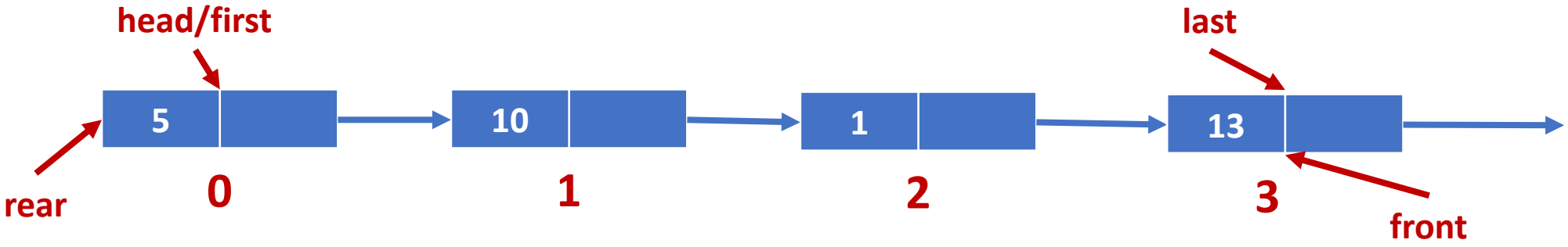
```
Queue<Integer> queue = new Queue<>();
```

Queue (LinkedList)



Where is the front and rear?

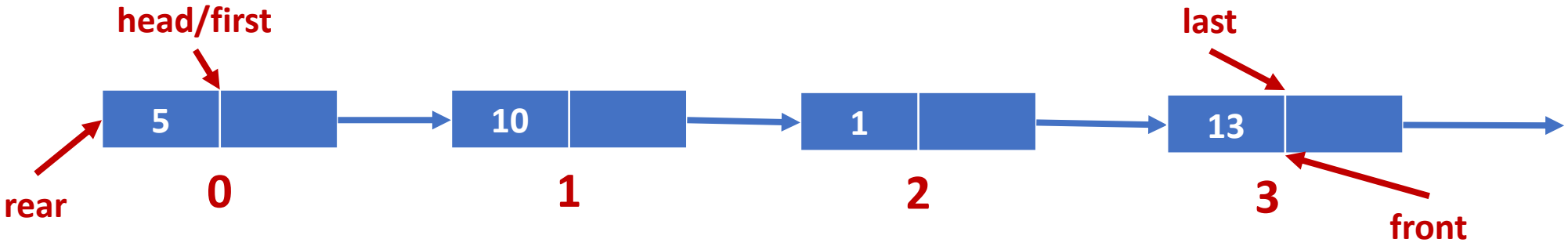
Queue (LinkedList)



Where is the front and rear?

Assume that rear = first and front = last

Queue (LinkedList)

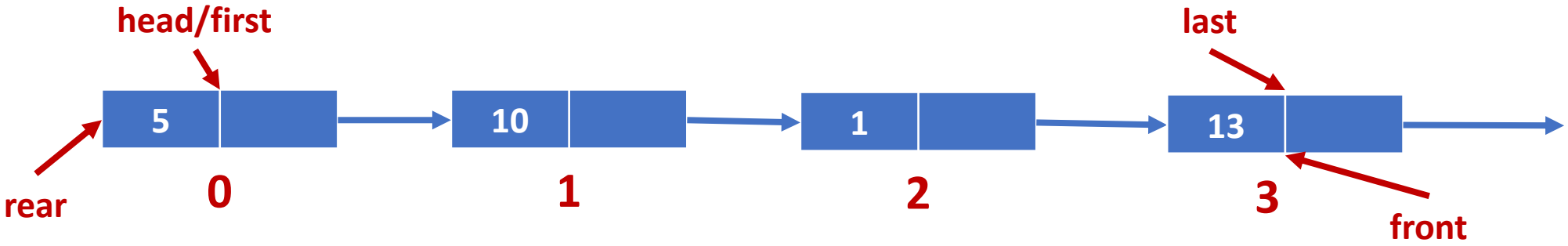


Where is the front and rear?

Assume that rear = first and front = last

enqueue() -> addFirst(); *// O(1)*

Queue (LinkedList)



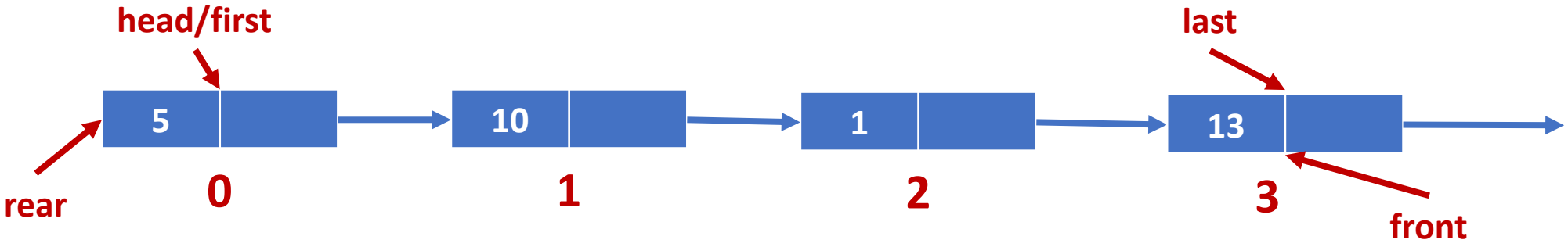
Where is the front and rear?

Assume that rear = first and front = last

enqueue() -> addFirst(); *// O(1)*

dequeue() -> removeLast(); *// O(n)*

Queue (LinkedList)



Where is the front and rear?

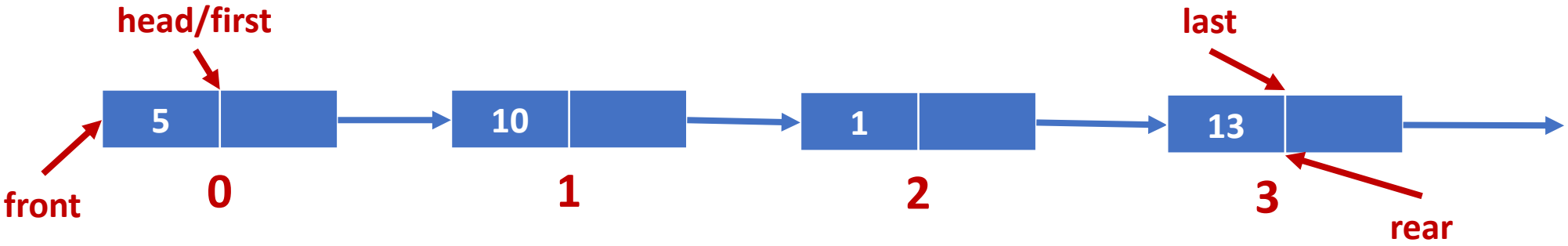
Assume that rear = first and front = last

enqueue() -> addFirst(); *// O(1)*

dequeue() -> removeLast(); *// O(n)*

This is valid because it's **FIFO** (First In First Out)

Queue (LinkedList)



Where is the front and rear?

Assume that rear = last and front = first

enqueue() -> addLast(); *// O(1)*

dequeue() -> removeFirst(); *// O(1)*

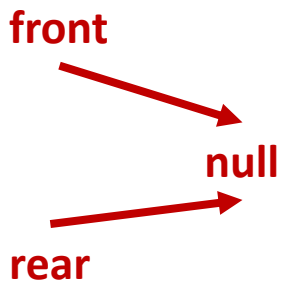
This is valid because it's **FIFO** (First In First Out)

Queue (LinkedList Implementation)

```
public class Node<T> {
    public T val;
    public Node<T> next;

    public Node(T val) {
        this(val, null);
    }

    public Node(T val, Node<T> next) {
        this.val = val;
        this.next = next;
    }
}
```

Queue (LinkedList Implementation)

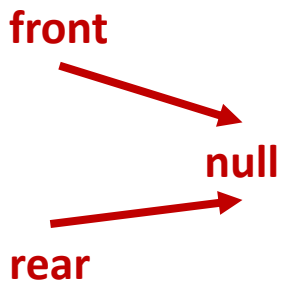
```
public class Queue<T> {
    int size = 0;
    Node<T> front;
    Node<T> rear;

    public Queue() {
        front = rear = null;
        size = 0;
    }
}
```



size = 0

Queue (LinkedList Implementation)

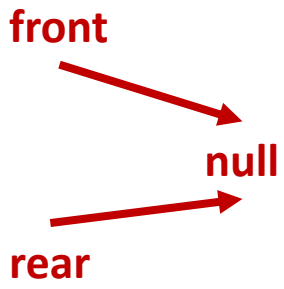
```
public boolean isEmpty() {
    return size == 0;
}

public int size() {
    return size;
}
```



size = 0

Queue (LinkedList Implementation)

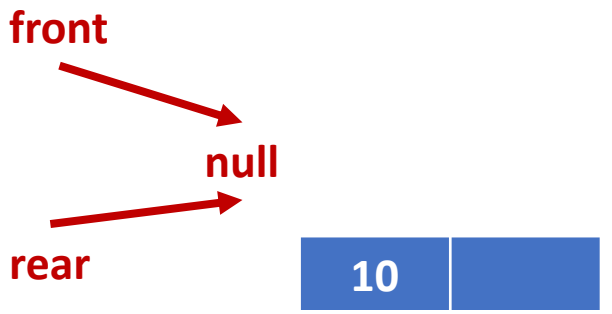
```
// Exactly like addLast
public void enqueue(T element) {
    Node<T> node = new Node<>(element);
    if (size == 0) {
        front = rear = node;
    } else {
        rear.next = node;
        rear = rear.next;
    }

    size++;
}
```



size = 0

Queue (LinkedList Implementation)

```
// Exactly like addLast
public void enqueue(T element) {
    Node<T> node = new Node<>(element);
    if (size == 0) {
        front = rear = node;
    } else {
        rear.next = node;
        rear = rear.next;
    }

    size++;
}
```

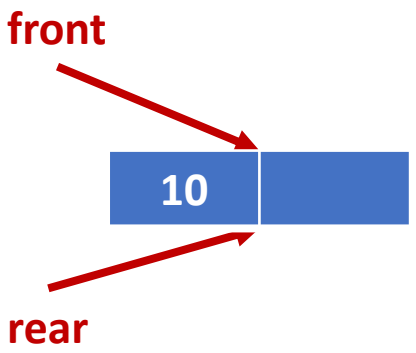


enqueue(10);

size = 0

Queue (LinkedList Implementation)

```
// Exactly like addLast
public void enqueue(T element) {
    Node<T> node = new Node<>(element);
    if (size == 0) {
        front = rear = node;
    } else {
        rear.next = node;
        rear = rear.next;
    }

    size++;
}
```

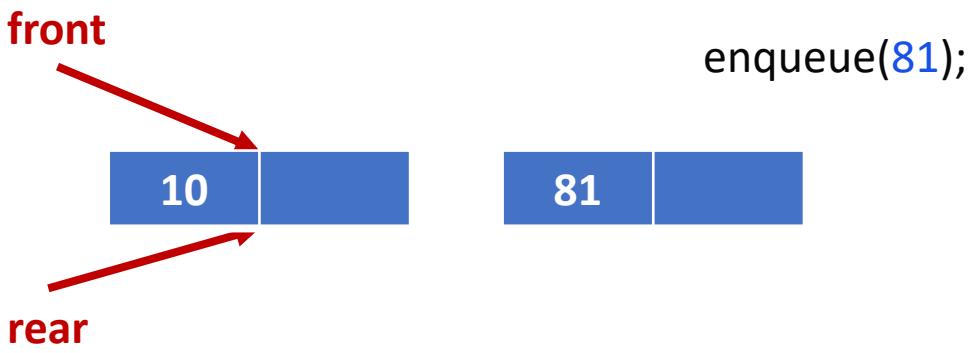


enqueue(10);

size = 1

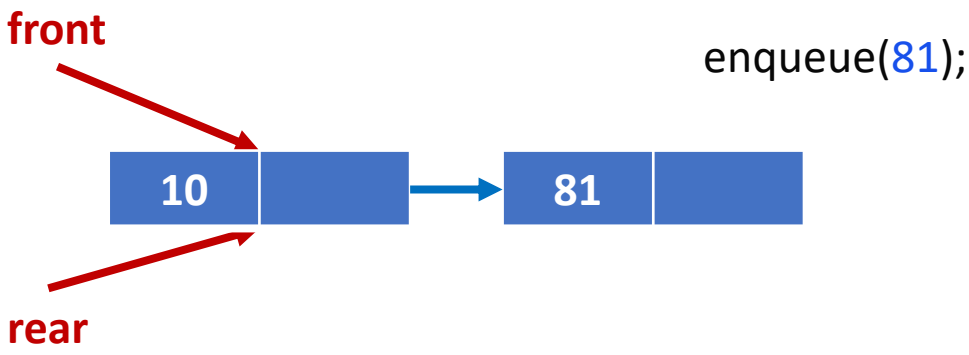
Queue (LinkedList Implementation)

```
// Exactly like addLast
public void enqueue(T element) {
    Node<T> node = new Node<>(element);
    if (size == 0) {
        front = rear = node;
    } else {
        rear.next = node;
        rear = rear.next;
    }

    size++;
}
```



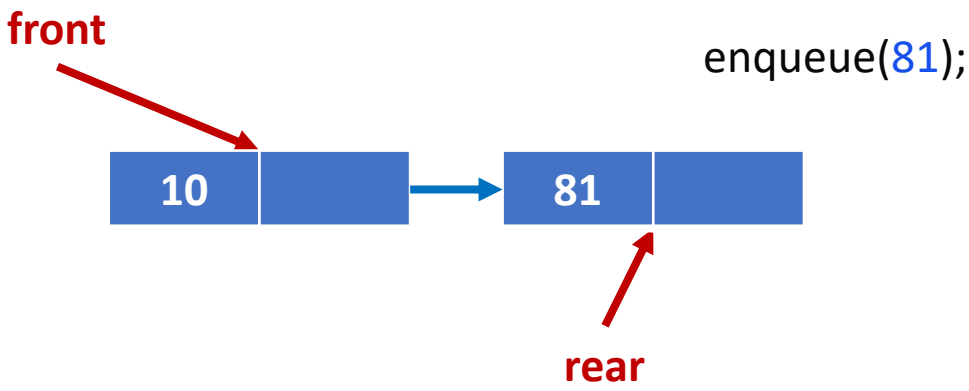
Queue (LinkedList Implementation)

```
// Exactly like addLast
public void enqueue(T element) {
    Node<T> node = new Node<>(element);
    if (size == 0) {
        front = rear = node;
    } else {
        rear.next = node;
        rear = rear.next;
    }

    size++;
}
```



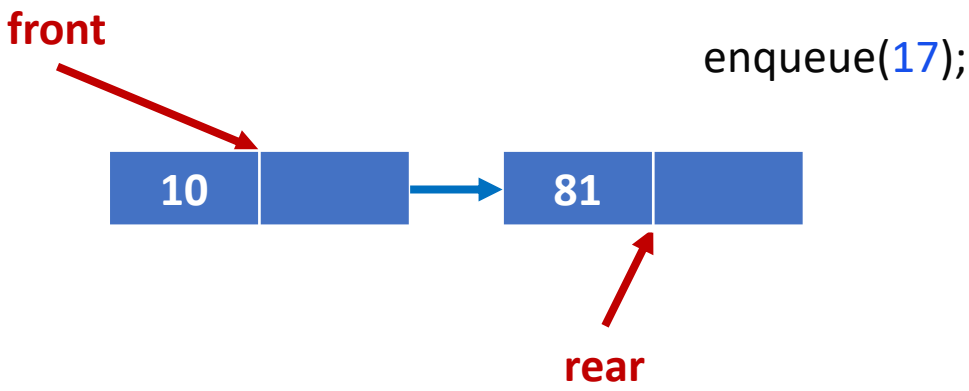
Queue (LinkedList Implementation)

```
// Exactly like addLast
public void enqueue(T element) {
    Node<T> node = new Node<>(element);
    if (size == 0) {
        front = rear = node;
    } else {
        rear.next = node;
        rear = rear.next;
    }

    size++;
}
```



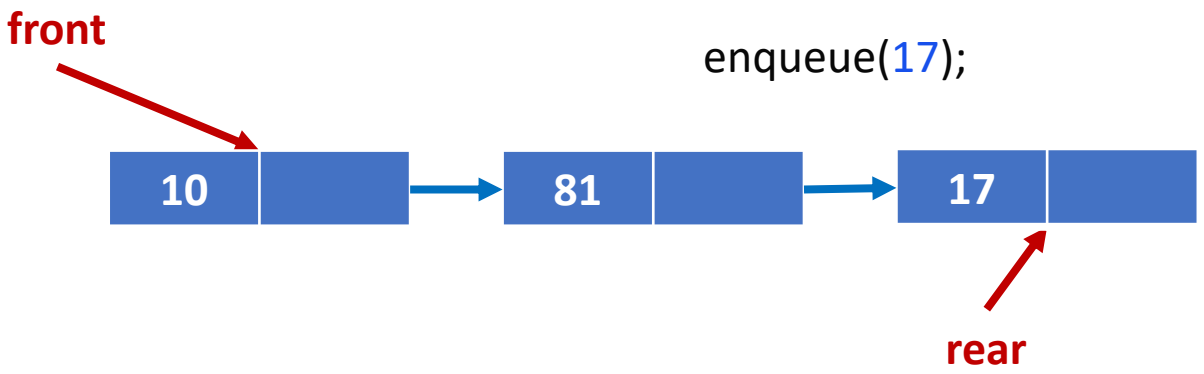
Queue (LinkedList Implementation)

```
// Exactly like addLast
public void enqueue(T element) {
    Node<T> node = new Node<>(element);
    if (size == 0) {
        front = rear = node;
    } else {
        rear.next = node;
        rear = rear.next;
    }

    size++;
}
```



Queue (LinkedList Implementation)

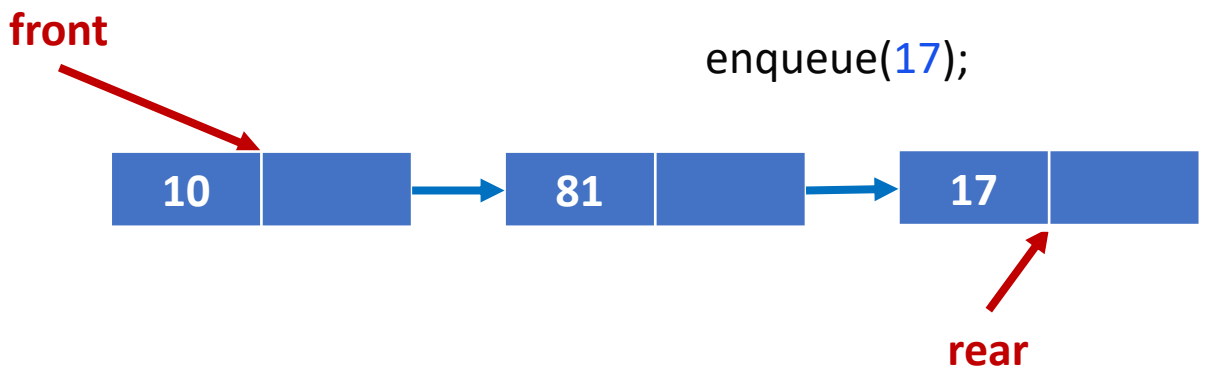
```
// Exactly like addLast
public void enqueue(T element) {
    Node<T> node = new Node<>(element);
    if (size == 0) {
        front = rear = node;
    } else {
        rear.next = node;
        rear = rear.next;
    }

    size++;
}
```



size = 3

Queue (LinkedList Implementation)

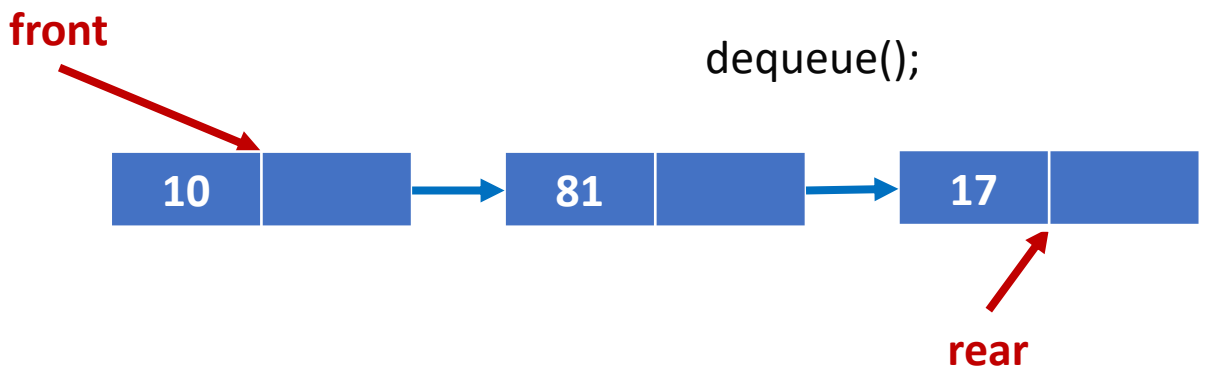
```
// Exactly like removeFirst
public T dequeue() {
   if (front == null) {
      return null;
   }

   T element = front.data;

   // Check if we have one element only
   if (front == rear) {
      front = rear = null;
   } else {
      front = front.next;
   }

   size--;
   return element;
}
```



size = 3

Queue (LinkedList Implementation)

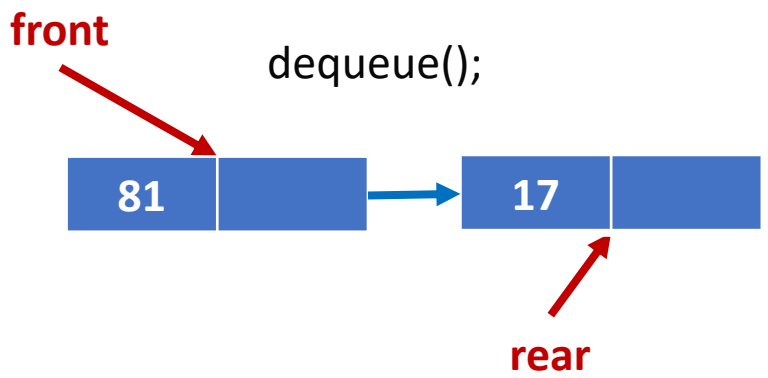
```
// Exactly like removeFirst
public T dequeue() {
   if (front == null) {
      return null;
   }

   T element = front.data;

   // Check if we have one element only
   if (front == rear) {
      front = rear = null;
   } else {
      front = front.next;
   }

   size--;
   return element;
}
```



size = 3

Queue (LinkedList Implementation)

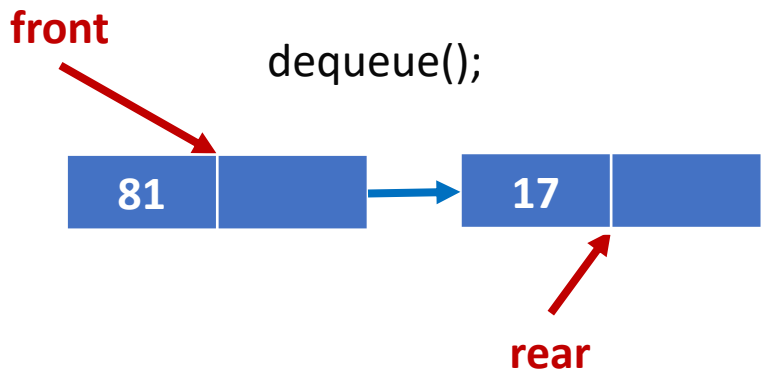
```
// Exactly like removeFirst
public T dequeue() {
   if (front == null) {
      return null;
   }

   T element = front.data;

   // Check if we have one element only
   if (front == rear) {
      front = rear = null;
   } else {
      front = front.next;
   }

   size--;
   return element;
}
```



size = 2

Queue (LinkedList Implementation)

```
public T front() {
   if (front == null) {
      return null;
   }

   return front.data;
}
```



Get element at the front of the queue!

size = 2

Queue (LinkedList Implementation)

How to loop on a queue?

Queue (LinkedList Implementation)

How to loop on queue?

```
while (!queue.isEmpty()) {
    int x = queue.dequeue();
    // Do something with x
}

// queue is empty here!
```

Queue (Exercise)

You have a queue of random integers!

```
Random random = new Random();
Queue<Integer> queue = new Queue<>();
for (int i = 0; i < 10; i++) {
    queue.enqueue(random.nextInt());
}
```

9	**587**
8	473
7	315
6	94
5	37
4	63
3	1025
2	1
1	77
0	10

Queue (Exercise)

You have a queue of random integers!

```
Random random = new Random();
Queue<Integer> queue = new Queue<>();
for (int i = 0; i < 10; i++) {
    queue.enqueue(random.nextInt());
}
```

9	587
8	473
7	315
6	94
5	37
4	63
3	1025
2	1
1	77
0	10

1. Find max in queue without destroying it!

```
int max = 0;
while (!queue.isEmpty()) {
    max = Math.max(max, queue.dequeue());
}

// queue is empty here! We don't want that!
```

Queue (Exercise)

You have a queue of random integers!

```
Random random = new Random();
Queue<Integer> queue = new Queue<>();
for (int i = 0; i < 10; i++) {
    queue.enqueue(random.nextInt());
}
```

1. Find max in queue without destroying it!
2. Print queue without destroying it!
3. Write a function to clear a queue

9	**587**
8	473
7	315
6	94
5	37
4	63
3	1025
2	1
1	77
0	10

Queue (Exercise)

Which data structure represents a waiting line and limits insertions to be made at the back and limits removals to be made from the front?

Fill the following table.

	Array	LinkedList
enqueue()	O(1)	O(1)
dequeue()		
front() or rear()		
Space efficiency		

Queue (Exercise)

Which data structure represents a waiting line and limits insertions to be made at the back and limits removals to be made from the front?

Fill the following table.

	Array	LinkedList
enqueue()	O(1)	O(1)
dequeue()		
front() or rear()		
Space efficiency		

Queue (Exercise)

Assume you have this code:

```
Queue<Integer> queue = new Queue<>();
queue.enqueue(13);
queue.enqueue(5);
queue.dequeue();
queue.enqueue(19);
queue.enqueue(17);
queue.dequeue();
queue.dequeue();
queue.enqueue(23);
```

What is the state of the queue after running the code:
1. Where are the front and rear pointing at?
2. What is the size?