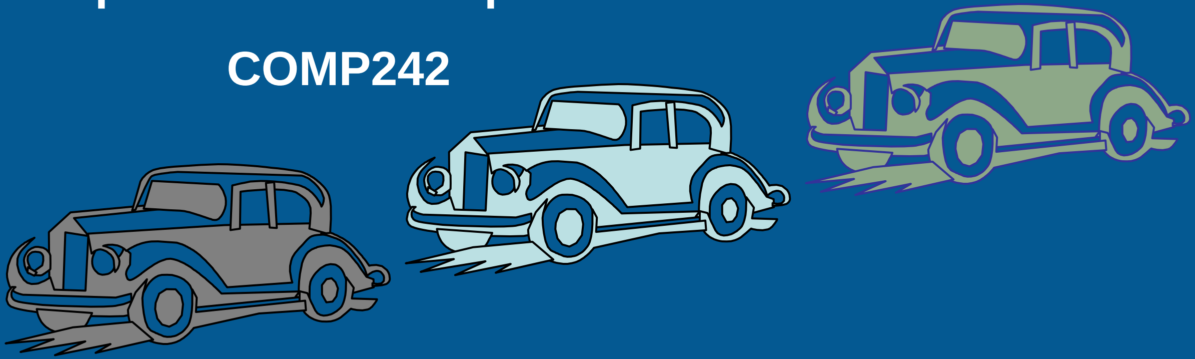
جامعة بيرزيت
BIRZEIT UNIVERSITY

Queues

Dr. Abdallah Karakra

Computer Science Department

COMP242



Wednesday, April 3, 2024

Review

A stack is a last in, first out (LIFO) data structure

– Items are removed from a stack in the reverse order from the way they were inserted

What about Queue?

Queue

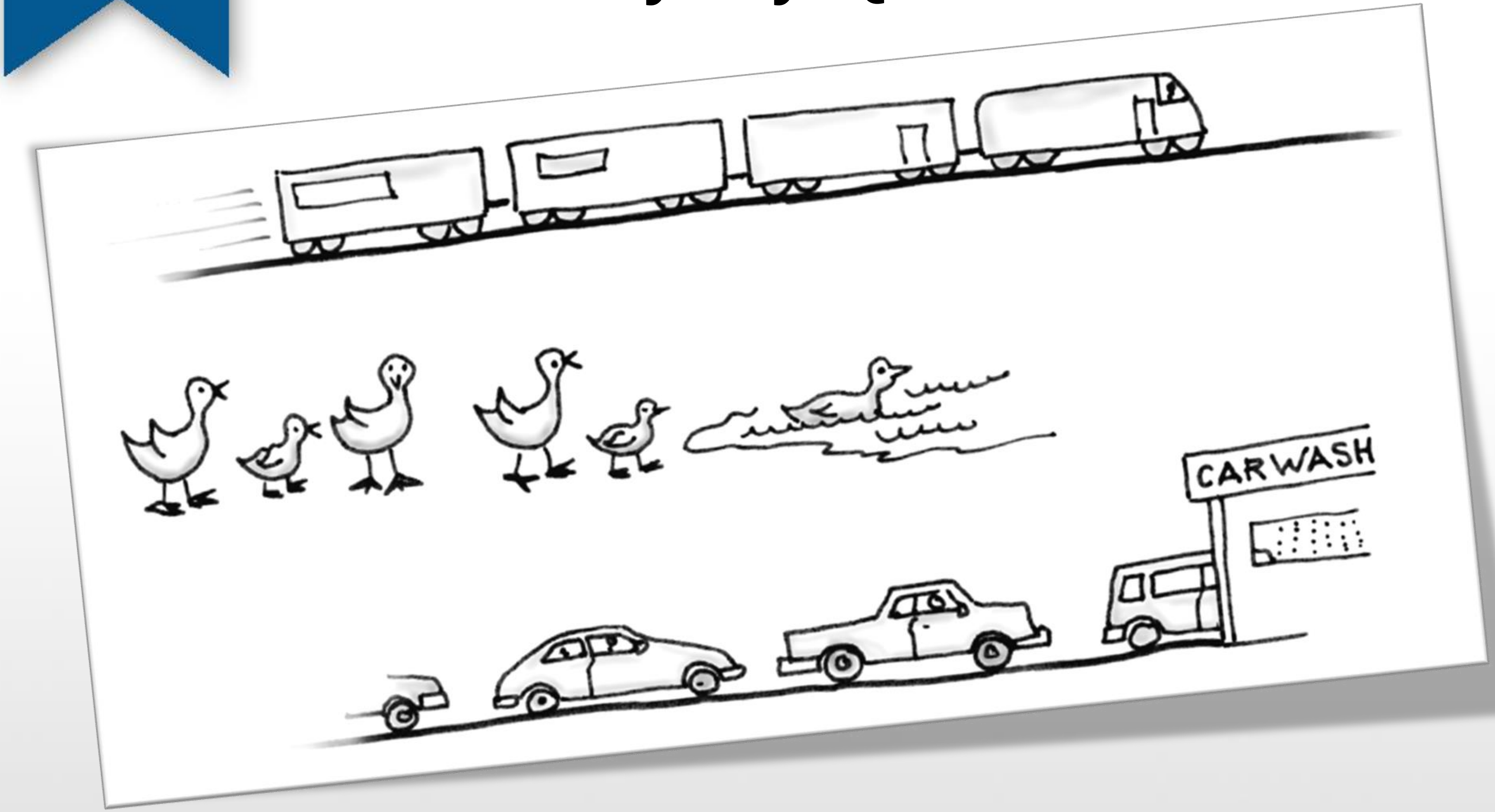
A queue is a first in, first out (FIFO) data structure

- Items are removed from a queue in the same order as they were inserted

(First item inserted is the first item removed; second inserted is second removed, third is third, etc.)

Rear of Queue
Front of Queue
next...
PAY HERE

Some Everyday Queues



Queues in computer science

❑ Operating systems:

- queue of print jobs to send to the printer

❑ Programming:

- modeling a line of customers or clients
- storing a queue of computations to be performed in order

❑ Real world examples:

- people on an escalator or waiting in a line
- cars at a gas station (or on an assembly line)

Queue Example

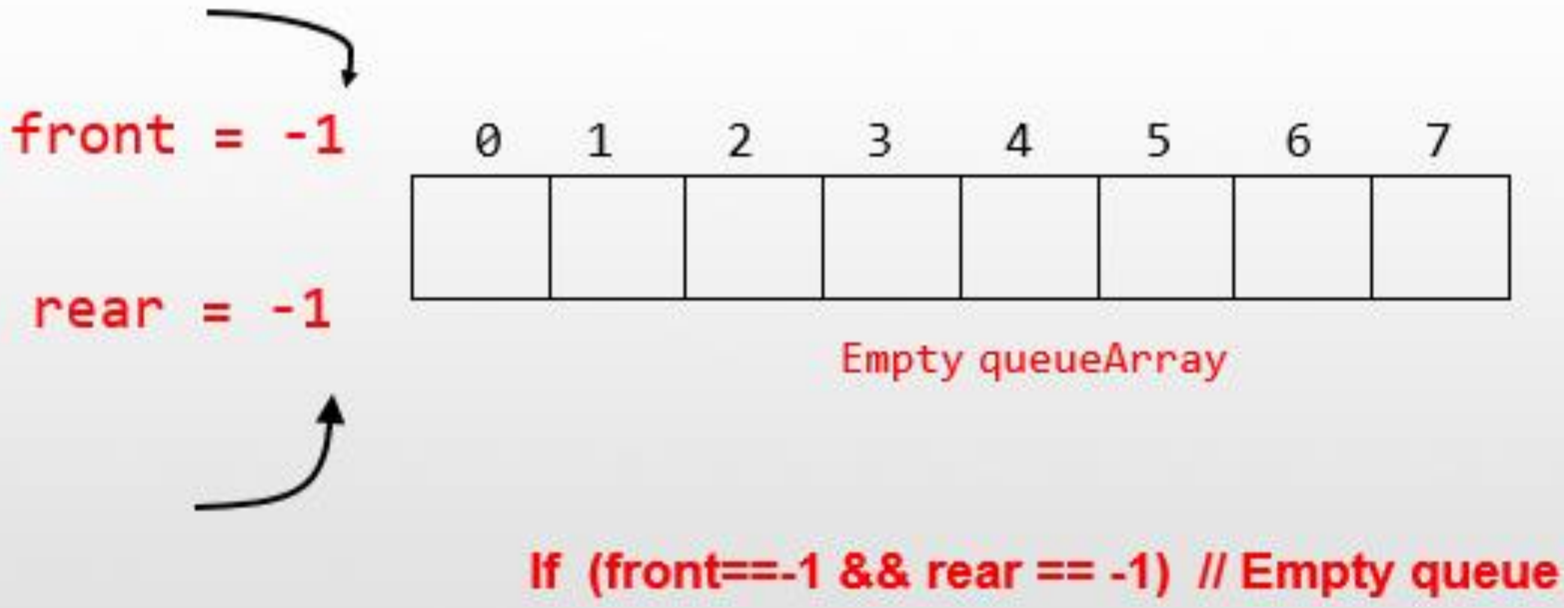
Operation	*Output*	*Q*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	*5*	(3)
enqueue(7)	–	(3, 7)
dequeue()	*3*	(7)
front()	*7*	(7)
dequeue()	*7*	()
dequeue()	*"error"*	()
isEmpty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	*9*	(7, 3, 5)

Operations

- **add** (enqueue): Add an element to the back.
- **remove** (dequeue): Remove the front element.
- **peek() or front()**: Examine the front element
- **isEmpty()**

Array implementation of queues

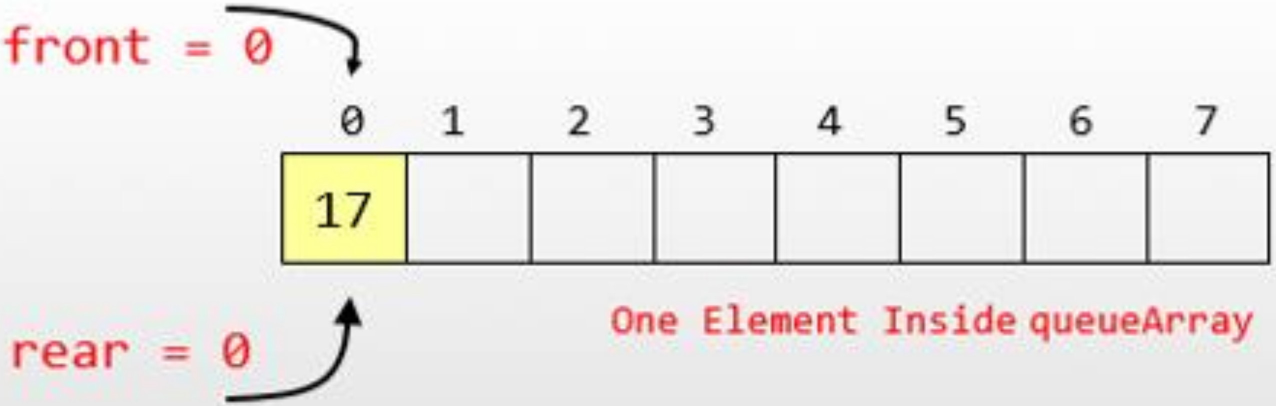
- □ A queue is a first in, first out (FIFO) data structure
- □ This is accomplished by inserting at one end (the rear) and deleting from the other (the front)



```
If  (front==-1 && rear == -1)  // Empty queue
```

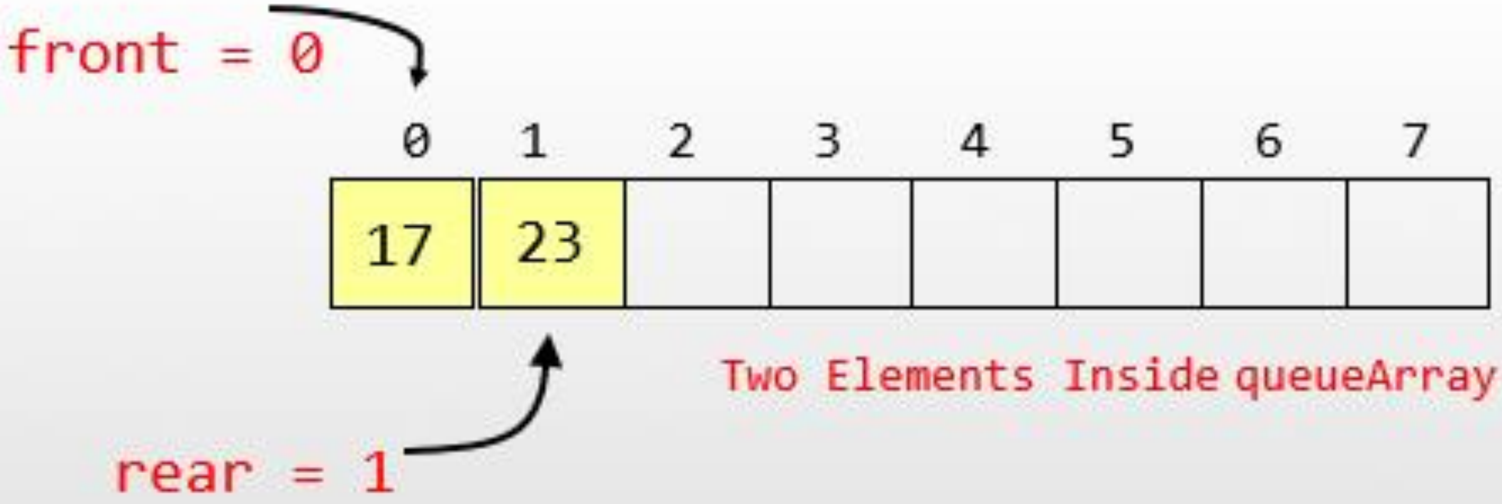
Array implementation of queues

- ❑ Enqueue (17)



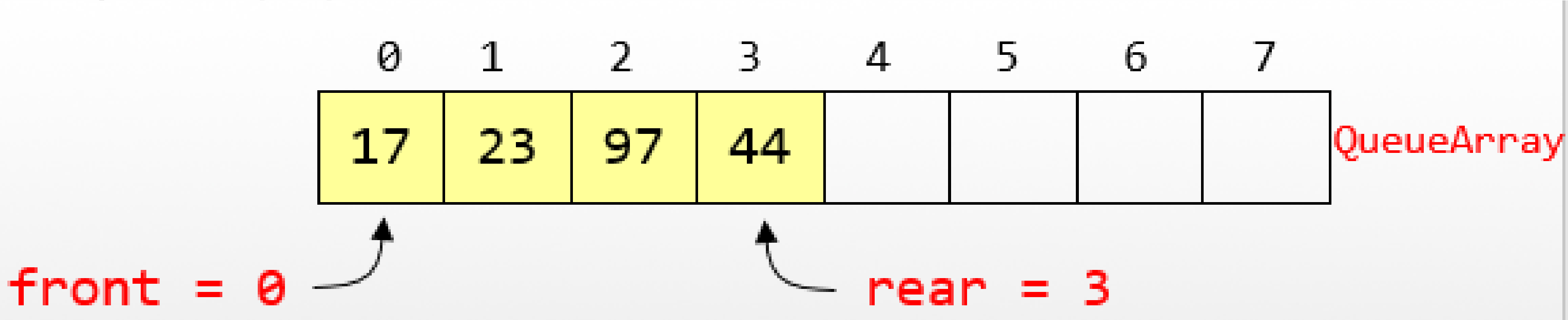
Array implementation of queues

- ❑ Enqueue (17)
- ❑ Enqueue (23)



Array implementation of queues

- ❑ Enqueue (17)
- ❑ Enqueue (23)
- ❑ Enqueue (97)
- ❑ Enqueue (44)



0	1	2	3	4	5	6	7
17	23	97	44				

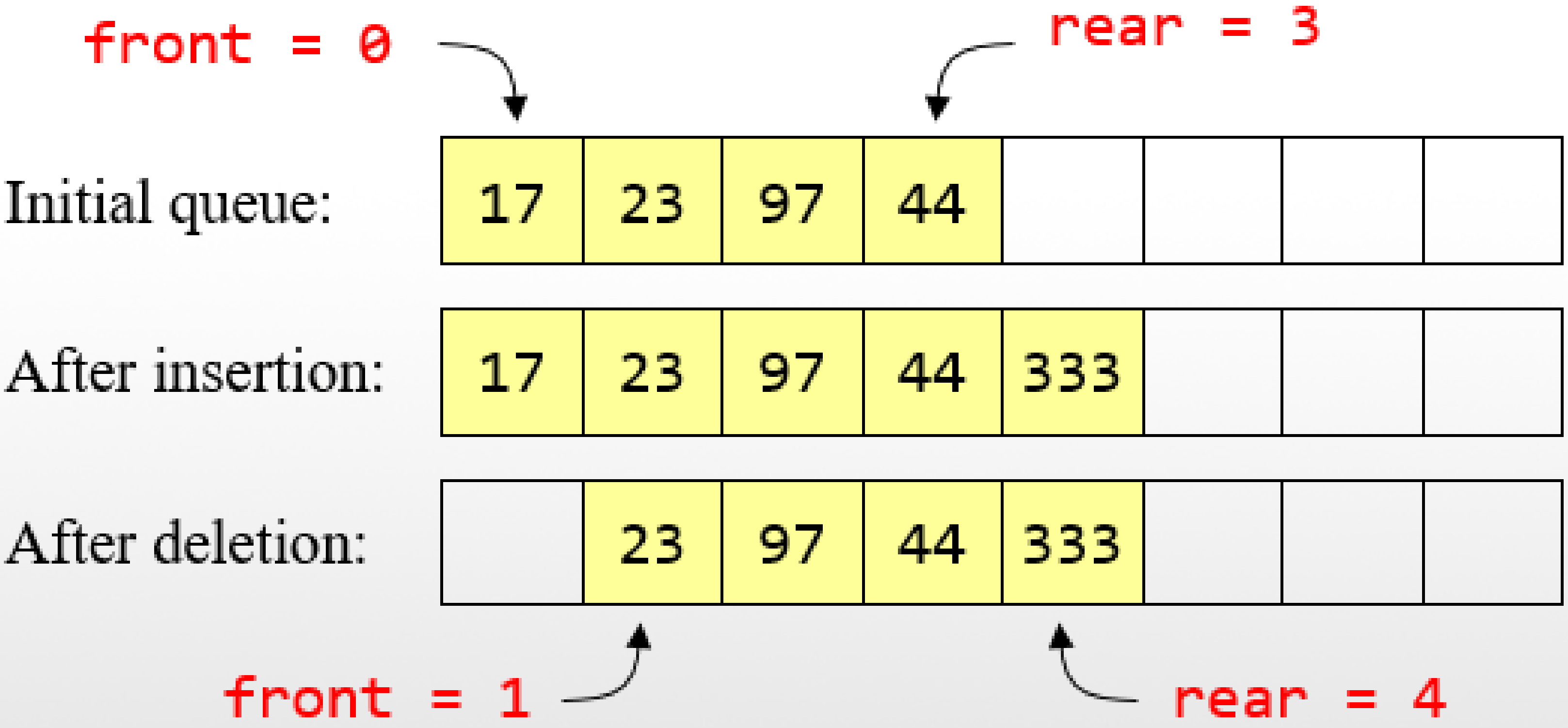
QueueArray

front = 0

rear = 3

- ❑ **To insert:** set rear to 4 , and put new element in location 4
- ❑ **To delete:** take element from location 0, and set front to 1

Array implementation of queues



front = 0

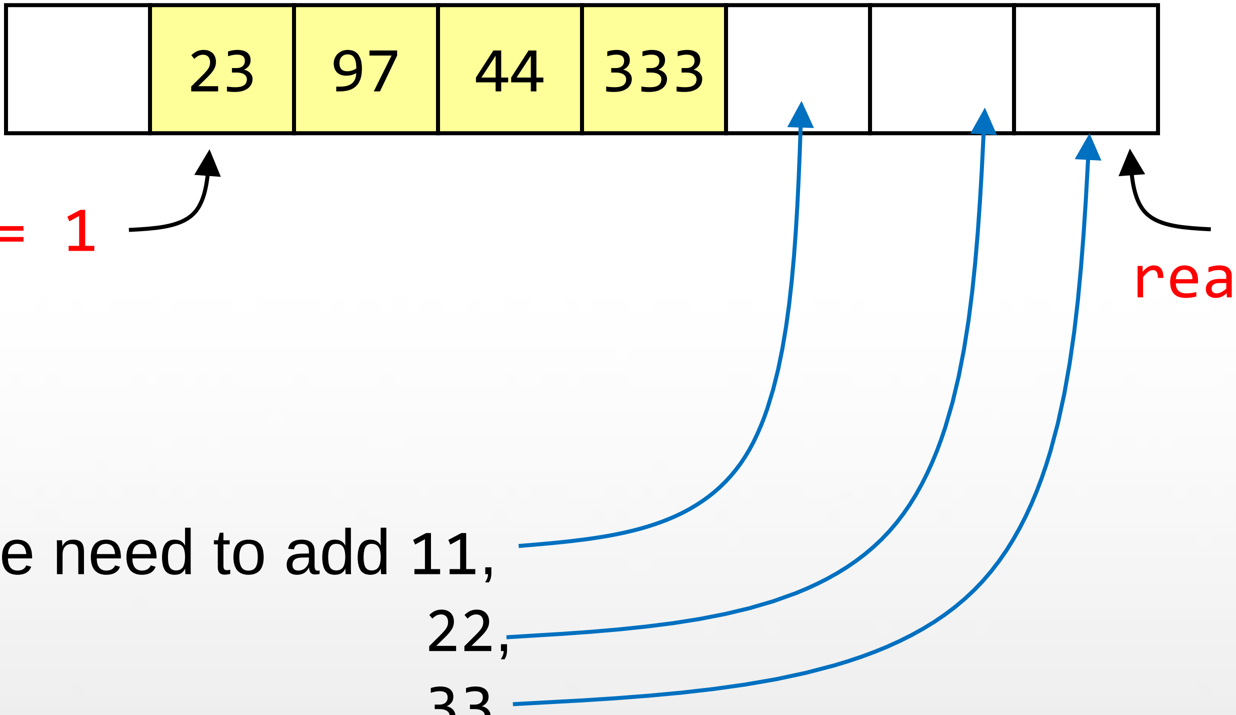
rear = 3

Initial queue:	17	23	97	44				
After insertion:	17	23	97	44	333			
After deletion:		23	97	44	333			

front = 1

rear = 4

Array implementation of queues



	23	97	44	333			

`front = 1`

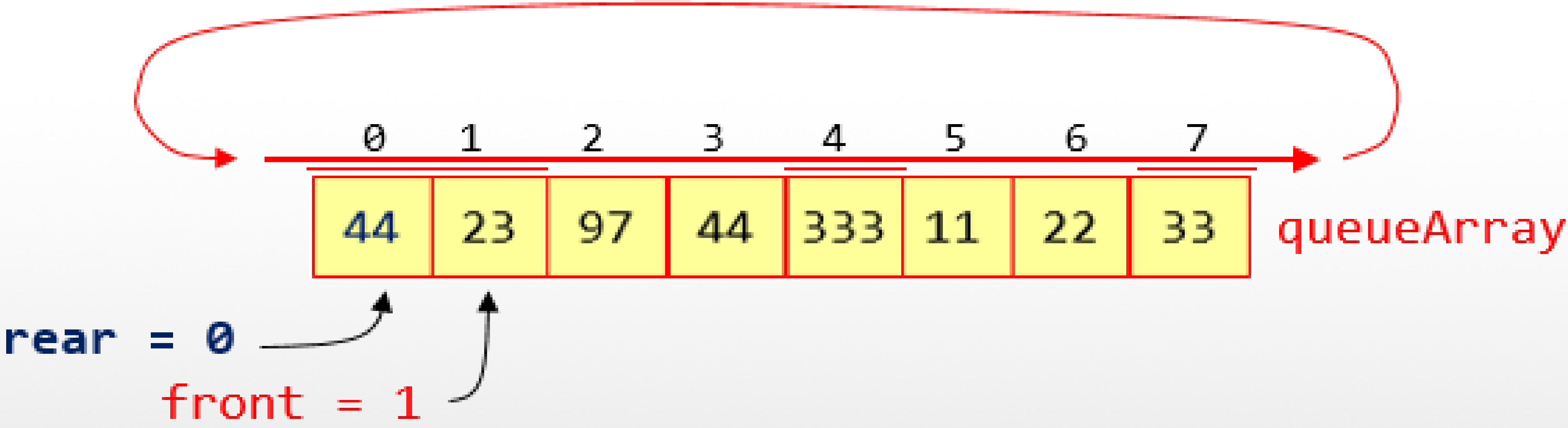
`rear = 7`

Suppose we need to add 11,
22,
33,
44 to this queue

- **Notice how the array contents “crawl” to the right as elements are inserted and deleted**
- **This will be a problem after a while!**

implementation of queues: Circular arrays

- We can treat the array holding the queue elements as circular (joined at the ends)



0	1	2	3	4	5	6	7
44	23	97	44	333	11	22	33

queueArray

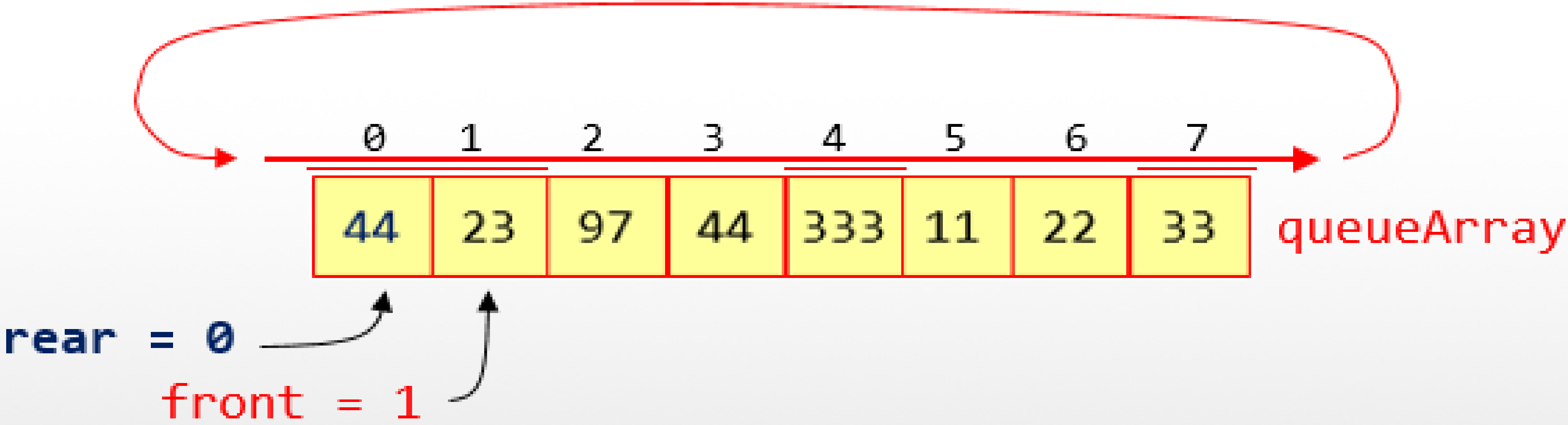
rear = 0

front = 1

- Elements were added to this queue in the order 11, 22, 33, **44** and will be removed in the same order
- Use: `front = (front + 1) % queueArray.length;`
  and: `rear = (rear + 1) % queueArray.length;`

implementation of queues: Circular arrays

- We can treat the array holding the queue elements as circular (joined at the ends)



If ((rear+1) % queueArray.length == front) // Full queue

Queue: Array-Based Implementation

Queue Class

```java
public class Queue {
   private int front;
   private int rear;
   private int maxSize; //queueArray size
   private Object [] queueArray;

   public Queue(int maxSize){

     front=rear=-1; //empty queue
     this.maxSize=maxSize;
     queueArray= new Object [maxSize];
   }
   /* Methods go here */
}
```

Queue: Array-Based Implementation

```
public void enQueue(Object element) {
   if (isFull())
      System.out.println("Queue is full");
   else if (isEmpty()) {
       front++;
       rear++;
       queueArray[rear] = element;
    }
    else
    {
         rear = (rear + 1) % maxSize;
         queueArray[rear] = element;
    }
}
```

Queue: Array-Based Implementation

```
public Object deQueue() {
   Object element = null;
   if (isEmpty())
       System.out.println("Queue is empty");
    else if (front == rear)
    {
        element = queueArray[front];
        front = rear = -1;
    }
    else {
       element = queueArray[front];
       front = (front + 1) % maxSize;
   }
   return element;
}
```

Queue: Array-Based Implementation

```
public boolean isEmpty(){// return true if the queue is empty

  return (front==-1 && rear==-1);

  }

public boolean isFull(){ // return true if the queue is full
  return ((rear+1)% maxSize == front);
}
```

Queue: Array-Based Implementation

```
public Object front(){//returns front

 if (isEmpty()) {
  System.out.println("Error: cannot return front from empty queue");
 return null;

 }

  return queueArray[front];

}
```

Queue implementation: H.W

You have one week to do the following

- ❑ Write a java function called **print**, to print the elements in queue from front to rear.

public void print();

Hint:

1. You have to find number of elements in queue
2. Use the formula (front+i) % maxSize   (circularly form)

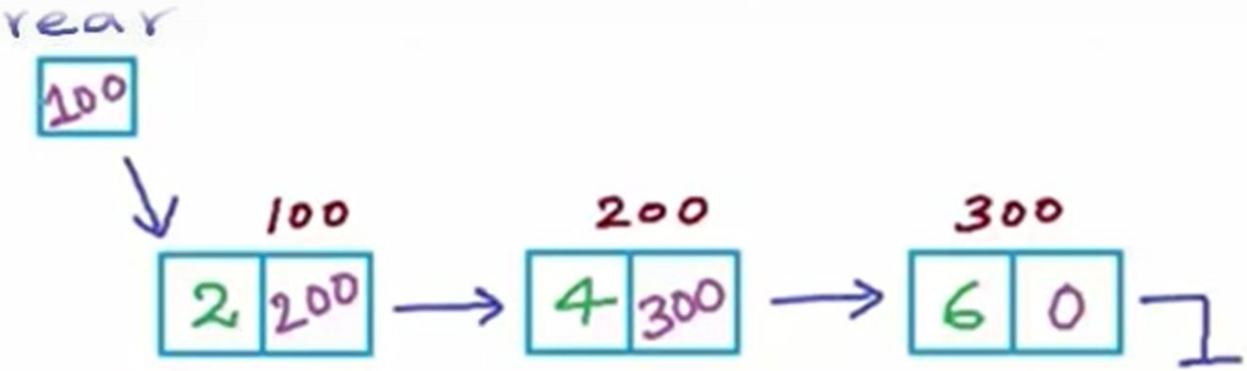
- ❑ Write a java function called **clear**, to clear the queue.

Public void clear ();

Queue- Linked list Implementation

Recall:

Queue is a list with the restriction that insertion can be performed at one end (rear) and deletion can be performed at other end (front).



Normal implementation of linked list

1. **Cost of insertion/ removal at head side is O(1) .**
2. **Cost of insertion/ removal at tail side is O(n) .**

Queue- Linked list Implementation

Recall:

```
enqueue
dequeue
front
isEmpty
```

Should take a constant time O(1)

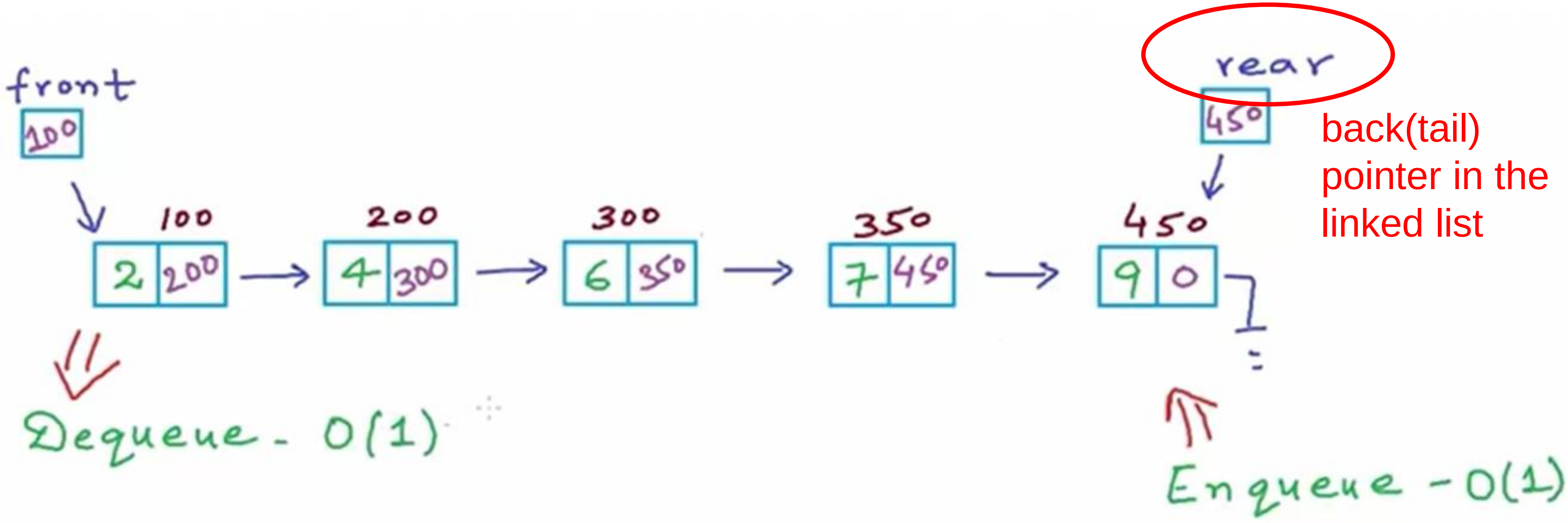
From the previous figures and in normal implementation of the linked list:

1. **Cost of insertion/ removal at head side is O(1) .**
2. **Cost of insertion/ removal at tail side is O(n) .**

The requirement both of these operations must take a constant time O(1)

Queue- Linked list Implementation

A list with the restriction that insertion can be performed at one end (rear) and deletion can be performed at other end (front).



back(tail) pointer in the linked list

Queue- Linked list Implementation

Node class

```
public class Node {
  public Object element;
  public Node next;

  public Node(Object element) {
    this(element, null);
  }

  public Node(Object element, Node next) {
    this.element = element;
    this.next = next;
  }
}
```

Queue- Linked list Implementation

Linked List class

```
public class LinkedListQueue {
  private Node front,rear;

  public LinkedListQueue () {
    front = rear = null;
  }

  /* Methods go here */
}
```

Queue implementation: H.W

You have one week to do the following

❑ Write a java code to implement all the following queue functions based on Link list.

```
enqueue
dequeue
front
isEmpty
```

Make sure the time for the above functions should be a constant time O(1)

Hint:

You have to use two references one points to the front of the list called front and the other points to the tail of the list called rear (or vice versa based on your implementation).

Extra Exercises

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

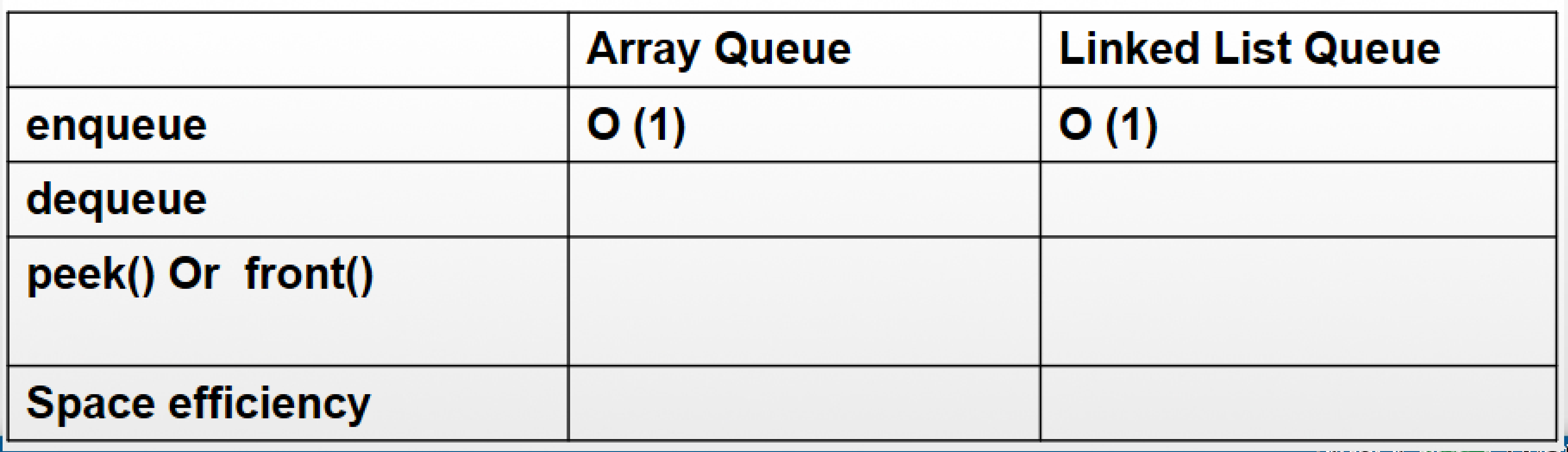
  a. Stack.

  b. Queue.

  c. Binary tree.

  d. Linked list.

- ❑ Fill the table below (Efficiency of the Queue Implementations)



	Array Queue	Linked List Queue
enqueue	O (1)	O (1)
dequeue		
peek() Or front()		
Space efficiency		

Extra Exercises

- ❑ Consider the following sequence of Queue operations:

enqueue(d), enqueue(h), dequeue(), enqueue(f), enqueue(s), dequeue(), dequeue(), enqueue(m).

Assume the Queue is initially empty, what is the sequence of dequeued values, and what would be the final state of the queue? (Identify which end is the front of the queue.)

Question?

“Success is the sum of small efforts, repeated day in and day out.”
Robert Collier

References:

1. Dr.David G. Sullivan Lecture Notes
2. Anwar Mamat Lecture Notes
3. Marty Stepp and Hélène Martin Lecture Notes