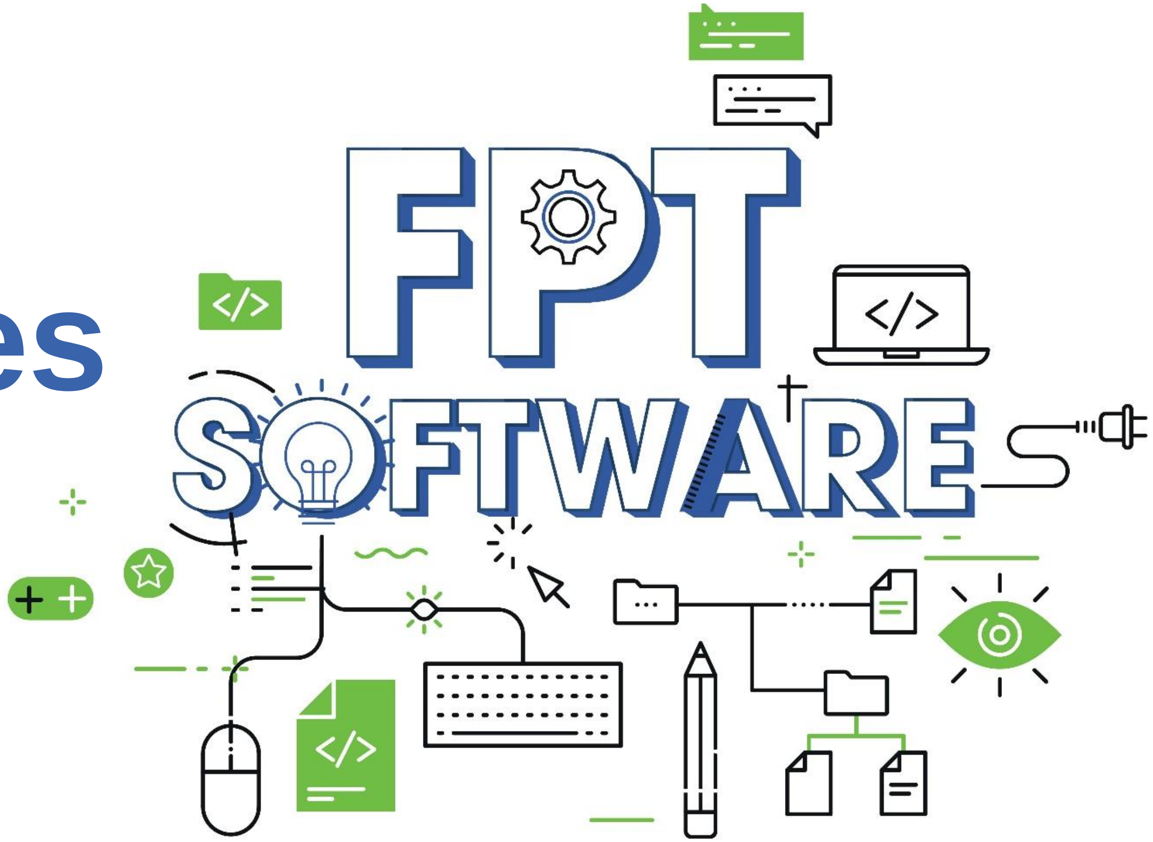
Stacks and Queues

Fsoft Academy



Lesson Objectives

- **Understand** the stacks and queues as abstract data types (ADTs).
- **Differentiate** between stacks and queues based on their LIFO (Last In, First Out) and FIFO (First In, First Out) principles, respectively.
- **Understand** different ways to implement stacks and queues, such as using arrays, linked lists, or specialized libraries.
- **Be able to Implement** basic operations like push, pop, peek, enqueue, dequeue, and isEmpty for both stacks and queues.

Agenda

1
- Stack

2
- Queue

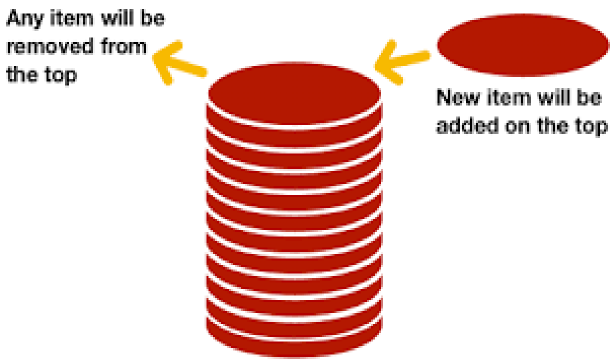
4
- Q&A

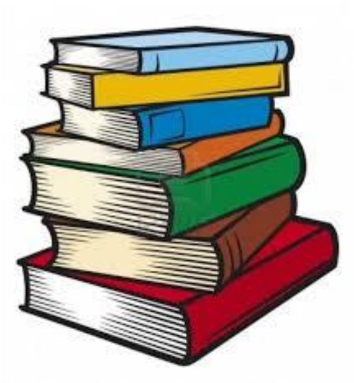
Section 1

STACK

Introduction

- **A stack** is an **Abstract Data Type** (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.





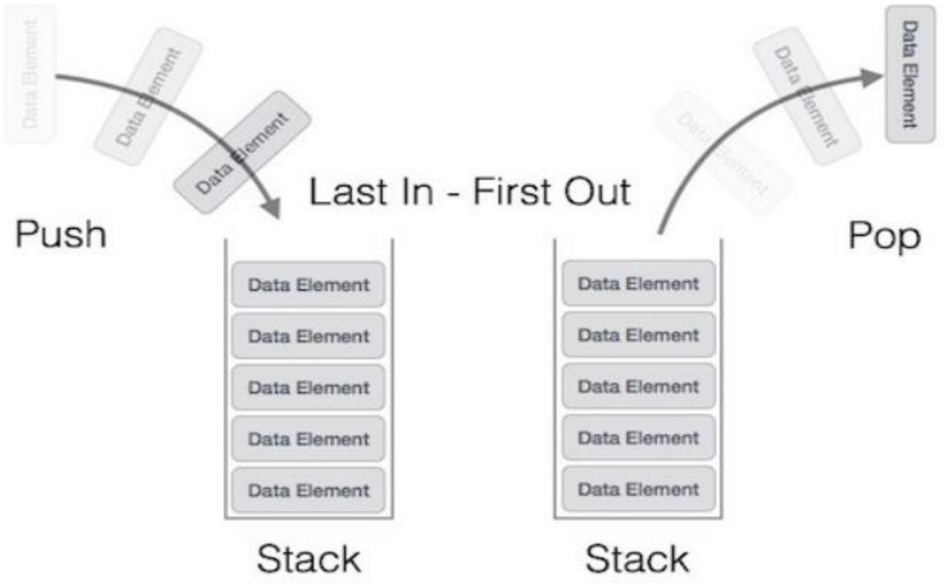
✓ A real-world stack allows operations at one end only. We can only access the top element of a stack.

✓ For example, we can place or remove a card or plate from the top of the stack only.

✓ This feature makes it **LIFO** data structure. LIFO stands for **Last-in-first-out**.

Stack Representation

- The element which is placed (*inserted* or *added*) last, is accessed first.
- In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.



✓ A stack can be implemented by means of *Array*, *Structure*, *Pointer*, and *Linked List*.

✓ Stack can either be a **fixed size** one or it may have a sense of dynamic resizing.

Stack Representation

- **There are two main ways to implement a stack –**
  - ✓Using array
  - ✓Using linked list
- **Example: define a stack using array**

```
public class MyStack {
   static final int MAXSIZE = 1000;
   int top;
   Integer stack[] = new Integer[MAXSIZE]; // Maximum size of Stack

   boolean isEmpty() {
       return (top < 0);
   }

   public MyStack() {
       top = -1;
   }
}
```

Basic Operations

- **push**() − pushing (storing) an element on the stack.
- **pop**() − removing (accessing) an element from the stack.
- **peek**() − get the top data element of the stack, without removing it.
- **isFull**() − check if stack is full.
- **isEmpty**() − check if stack is empty.

Notes: At all times, we maintain a pointer to the last pushed data on the stack. As this pointer always represents the top of the stack, hence named top. The top pointer provides top value of the stack without actually removing it.

Basic Operations

- **peek()** method:
- Algorithm of **peek**() function

```
begin procedure peek
  if top top less than 0
    return null
  return
    stack[top]
end procedure
```

- Code:

```java
public Integer peek() {
  if (top < 0) {
    System.out.println("Could not peek data:
                                Stack is empty!");
    return null;
  } else {
    int value = stack[top];
    return value;
  }
}
```

Basic Operations

- **isFull()** method:
- Algorithm of **isFull**() function

```
begin procedure isFull
   if top equals to MAXSIZE-1
      return true
   else
      return false
   endif
end procedure
```

- Code:

```
public boolean isFull() {
   if(top == MAXSIZE-1)
      return true;
   else
      return false;
}
```

Basic Operations

- **isEmpty()** method:
- Algorithm of **isEmpty**() function

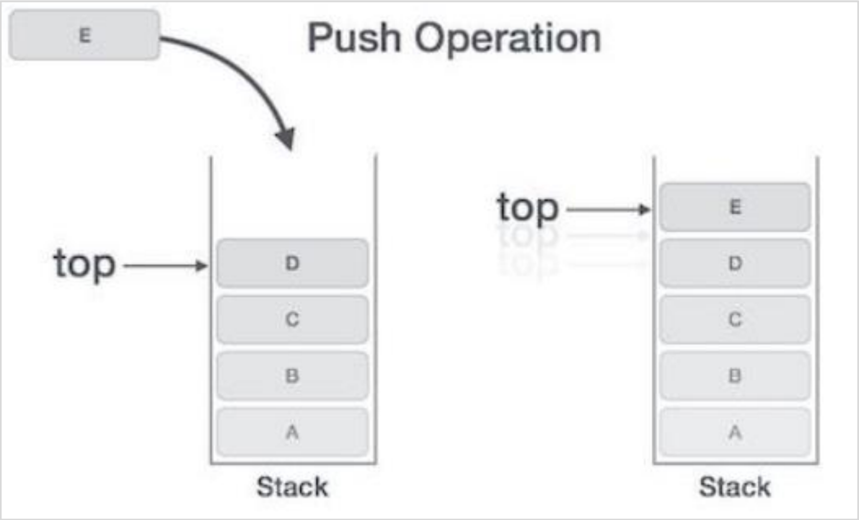
```
begin procedure isEmpty
   if top less than 0
      return true
   else
      return false
   endif
end procedure
```

- Code:

```java
public boolean isEmpty() {
   if(top == -1)
      return true;
   else
      return false;
}
```

Basic Operations

- **Push Operation**: The process of putting a *new data element onto stack* is known as a Push Operation.
  - ✓ **Step 1**: Checks if the stack is full.
  - ✓ **Step 2**: If the stack is full, produces an error and exit.
  - ✓ **Step 3**: If the stack is not full, increments top to point next empty space.
  - ✓ **Step 4**: Adds data element to the stack location, where top is pointing.
  - ✓ **Step 5**: Returns success.



Basic Operations

- **push()** method:
  - Algorithm of push() function

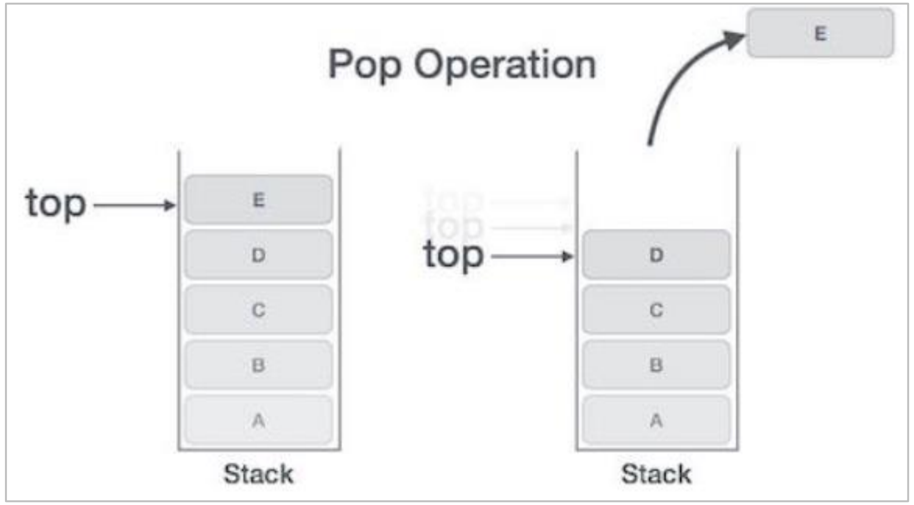
```
begin procedure push: stack, data
   if stack is full
      return false
   else
      top ← top + 1
      stack[top] ← data
      return true
   endif
end procedure
```

- Code:

```java
public boolean push(int data) {
  if (!isFull()) {
    stack[++top] = data;
    return true;
  } else {
    System.out.println("Could not insert data: Stack is full!");
    return false;
  }
}
```

Basic Operations

- **Pop Operation:** Accessing the content while *removing it from the stack*, is known as a <u>Pop Operation</u>.
  - ✓ **Step 1** − Checks if the stack is empty.
  - ✓ **Step 2** − If the stack is empty, produces an error and exit.
  - ✓ **Step 3** − If the stack is not empty, accesses the data element at which top is pointing.
  - ✓ **Step 4** − Decreases the value of top by 1.
  - ✓ **Step 5** − Returns success.



Basic Operations

- **pop()** method:
- Algorithm of **pop**() function

```
begin procedure pop: stack
   if stack is empty
      return null
   else
      data ← stack[top]
      top ← top - 1
      return data
   endif
end procedure
```

- Code:

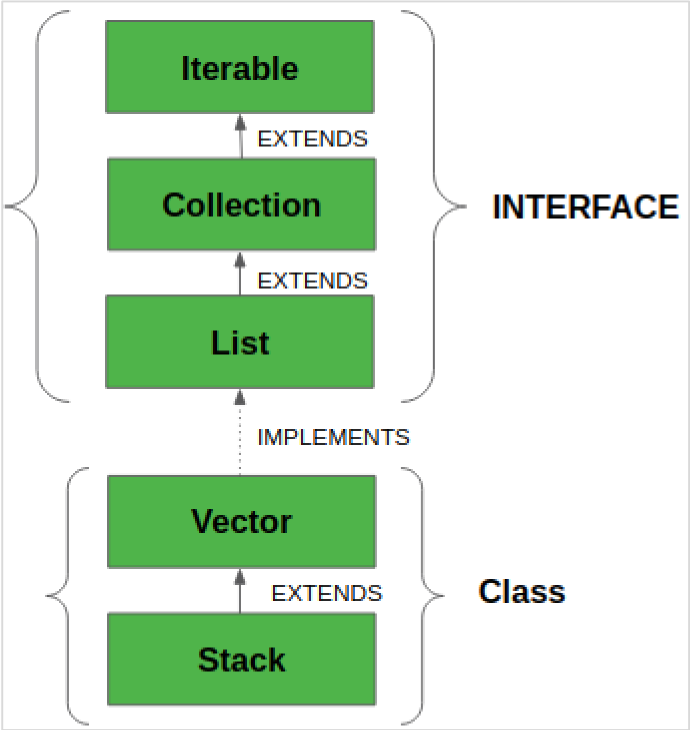
```java
public Integer pop() {
  if (!isEmpty()) {
    int value = stack[top--];
    return value;
  } else {
    System.out.println("Could not retrieve data:
                                Stack is empty!");
    return null;
  }
}
```

Stack

- Code demo!

Stack Class in Java

- **Java Collection framework** provides a **Stack class** that models and implements a Stack data structure.
- The class is based on the basic principle of **last-in-first-out**.
- In addition to the basic *push* and *pop operations*, the class provides *three more functions of* ***empty****,* ***search****, and* ***peek****.* The class can also be said to **extend Vector** and treats the class as a stack with the five mentioned functions.



Stack Class in Java

- **Example:**

```java
public class StackDemo {
    public static void main(String args[]) {
        // Creating an empty Stack
        Stack<Integer> stack = new Stack<Integer>();
        // Use add() method to add elements
        stack.push(10);
        stack.push(15);
        stack.push(30);
        stack.push(20);
        stack.push(5);
        // Displaying the Stack
        System.out.println("Initial Stack: " + stack);
        // Removing elements using pop() method
        System.out.println("Popped element: " + stack.pop());
        System.out.println("Popped element: " + stack.pop());

        // Fetching the element at the head of the Stack
        System.out.println("The element at the top of the stack is: " + stack.peek());
        System.out.println("Seaching the element in stack:"+ stack.search(15));
        System.out.println("Seaching the element in stack:"+ stack.search(28));

        // Displaying the Stack after pop operation
        System.out.println("Stack after pop operation " + stack);
    }
}
```

- **Output:**

```
Initial Stack: [10, 15, 30, 20, 5]
Popped element: 5
Popped element: 20
The element at the top of the stack is: 30
Seaching the element in stack:2
Seaching the element in stack:-1
Stack after pop operation [10, 15, 30]
```

Section 2

QUEUES

Introduction

- **Queue** is an **abstract data structure**, somewhat *similar to Stacks*.
- **Unlike stacks**, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue).

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

Introduction

- A real-world example of the queue can be a *single-lane one-way road, where the vehicle enters first, exits first*.
- More real-world examples can be seen as queues at the ticket windows and bus-stops.

Notes: As in stacks, a queue can also be implemented using ***Arrays****,* ***Linked-lists****,* ***Pointers*** *and* ***Structures****. In the lesson, we shall implement queues using one-dimensional array.*

Queue Representation

- **Array Representation of Queue:** Like stacks, Queues can also be represented in an array: In this representation, the Queue is implemented using the array. Variables used in this case are
  - ✓ **Queue:** the name of the array storing queue elements.
  - ✓ **Front**: the index where the first element is stored in the array representing the queue.
  - ✓ **Rear:** the index where the last element is stored in an array representing the queue.
- **Example:**

```java
public class MyQueue {
    private int front, rear, capacity, currentSize;
    private Integer queue[];

    public MyQueue(int capacity) {
        this.capacity = capacity;
        front = 0;
        rear = -1;
        currentSize = 0;
        queue = new Integer[this.capacity];
    }
}
```

Basic Operations

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

Notes: In queue, we always dequeue (or access) data, pointed by front pointer and while enqueing (or storing) data in the queue we take help of rear pointer.

Basic Operations

- **peek**() method: This function helps to see the data at the front of the queue. The algorithm of peek() function is as follows

- **Algorithm of peek() function:**

```
begin procedure peek
 if queue is empty
  return null
 else
  return queue[front]
 endif
end procedure
```

❖ **Code:**

```java
public Integer peek() {
   if(isEmpty()) {
      System.out.println("Queue is empty!");
      return null;
   }else {
      return queue[front];
   }
}
```

Basic Operations

- **isFull()** method: As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full.

- **Algorithm of isFull() function:**

```
begin procedure isFull
   if currentSize equals to MAXSIZE
      return true
   else
      return false
   endif
end procedure
```

❖ **Code:**

```java
public boolean isFull() {
  if (currentSize == capacity) {
     return true;
  }

  return false;
}
```

Basic Operations

- **isEmpty()** method: If the value of front is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.
- **Algorithm of isEmpty() function**

```
begin procedure isEmpty

 if currentSize equals 0
        return true
 return false
 endif

end procedure
```

❖ **Code:**

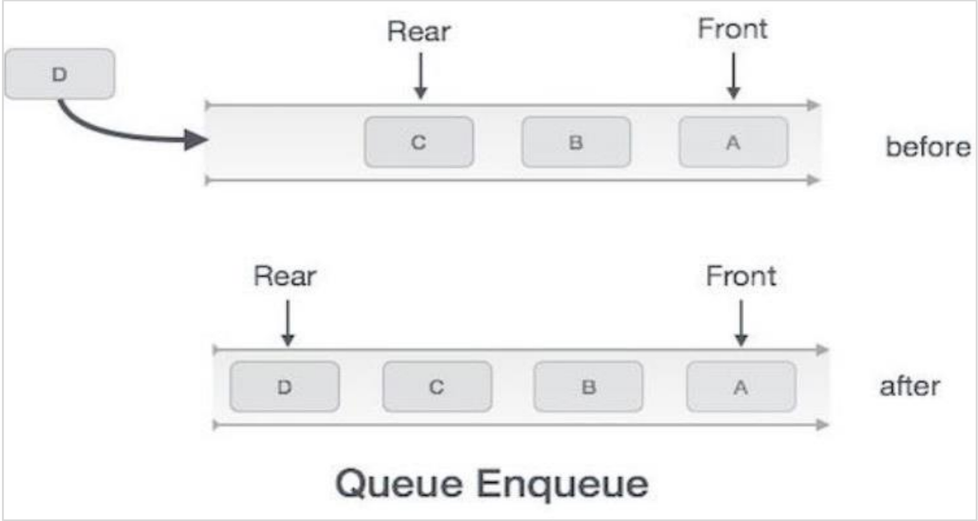
```
public boolean isEmpty() {
  if (currentSize == 0) {
     return true;
  }

  return false;

}
```

Basic Operations

- **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, produce overflow error and exit.
  - ✓ **Step 3** − If the queue is not full, increment rear pointer to point the next empty space.
  - ✓ **Step 4** − Add data element to the queue location, where the rear is pointing.
  - ✓ **Step 5** − return success.



Basic Operations

- **enqueue()** method: If the value of front is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.
- **Algorithm of enqueue() function**

```
procedure enqueue(data)
   if queue is full
      return overflow
   endif

   rear ← rear + 1
   queue[rear] ← data
   currentSize ← currentSize + 1
   return true
end procedure
```

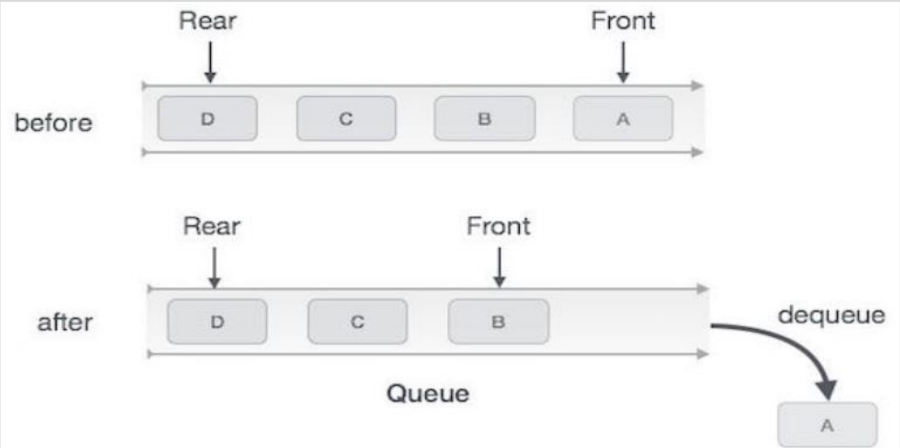
❖ **Code:**

```java
public boolean enqueue(int data) {
   /* Checks whether the queue is full or not */
   if (isFull()) {
      System.out.println("Queue is full!");
      return true;
   }

   /* increment rear then insert element to queue */
   queue[++rear] = data;
   currentSize++;
   System.out.println("Item added to queue: " + data);
   return true;
}
```

Basic Operations

- **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.



Queue Dequeue

Basic Operations

- **dequeue()** method:
- **Algorithm of dequeue() function**

```
procedure dequeue
   if queue is empty
      return underflow
   end if

   data = queue[front]
   front ← front + 1
   currentSize--;

   return data
end procedure
```

- **Code:**

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

   int value = queue[front++];
   currentSize--;

   return value;
}
```

Queue

- **Code demo!**

```
public static void main(String[] args) {
MyQueue myQueue = new MyQueue(5);

myQueue.enqueue(20);// rear = 0
myQueue.enqueue(8); // rear = 1
myQueue.enqueue(30);// rear = 2
myQueue.enqueue(16);// rear = 3
myQueue.enqueue(25);// rear = 4
myQueue.enqueue(88);// rear = 5

System.out.println("Dequeue: "+ myQueue.dequeue());
System.out.println("Retrive: "+ myQueue.peek());
System.out.println("Dequeue: "+ myQueue.dequeue());
System.out.println("Dequeue: "+ myQueue.dequeue());

System.out.print("Display Queue:");
myQueue.print();
}
```

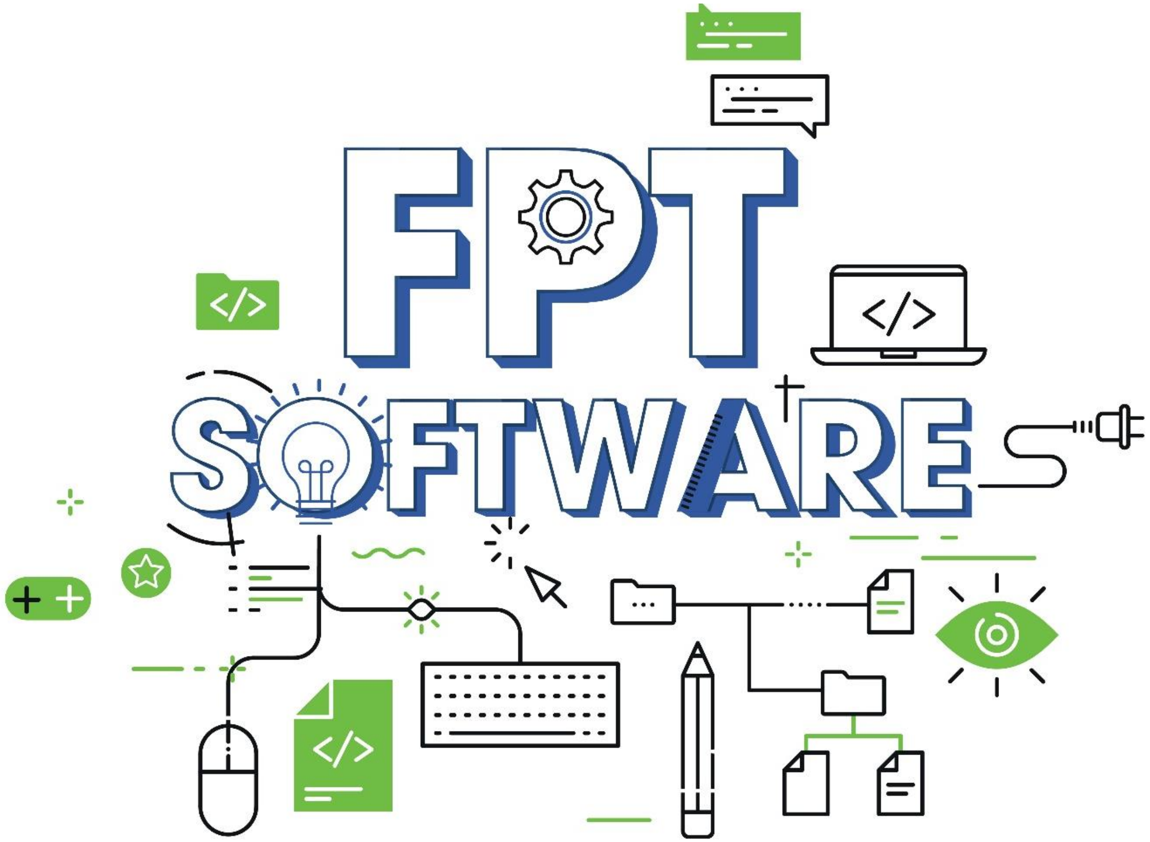
- **Output:**

```
Item added to queue: 20
Item added to queue: 8
Item added to queue: 30
Item added to queue: 16
Item added to queue: 25
Queue is full!
Dequeue: 20
Retrive: 8
Dequeue: 8
Dequeue: 30
Display Queue:16 25
```

Summary

- Stack
- Queue
- Q&A

Q & A



THANK YOU!