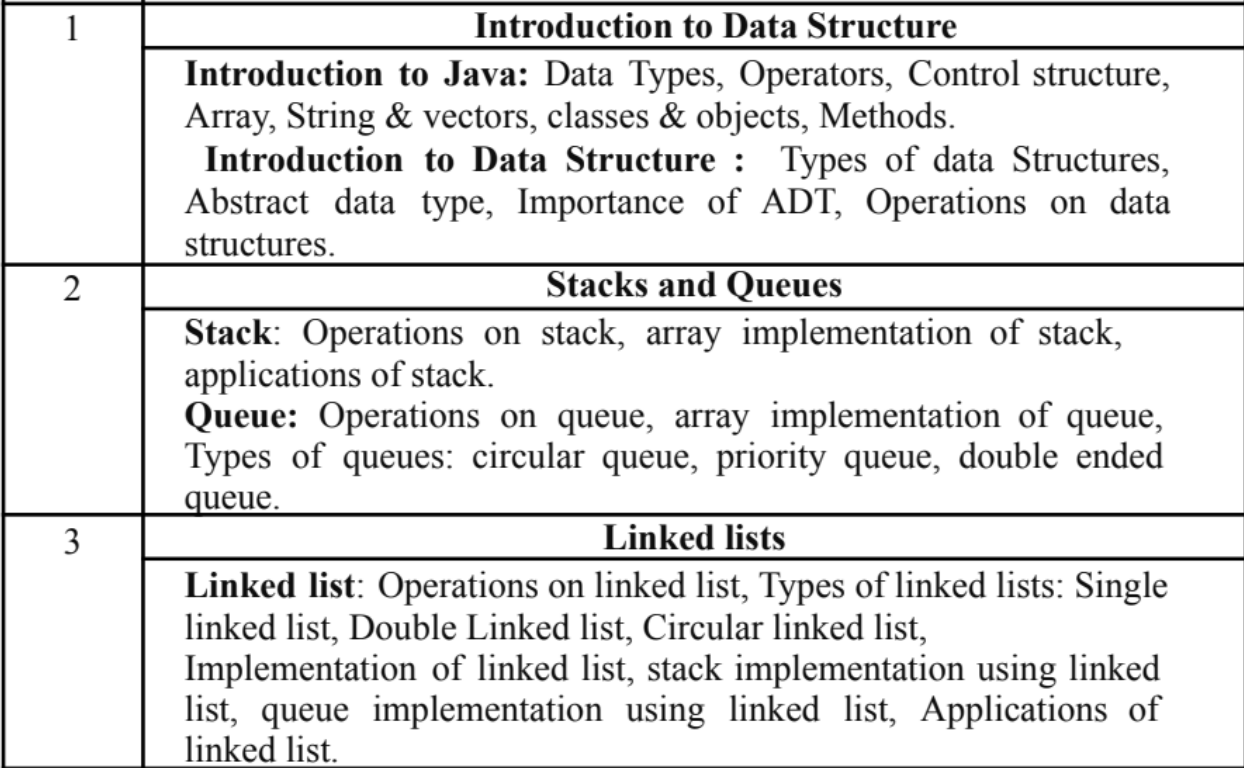
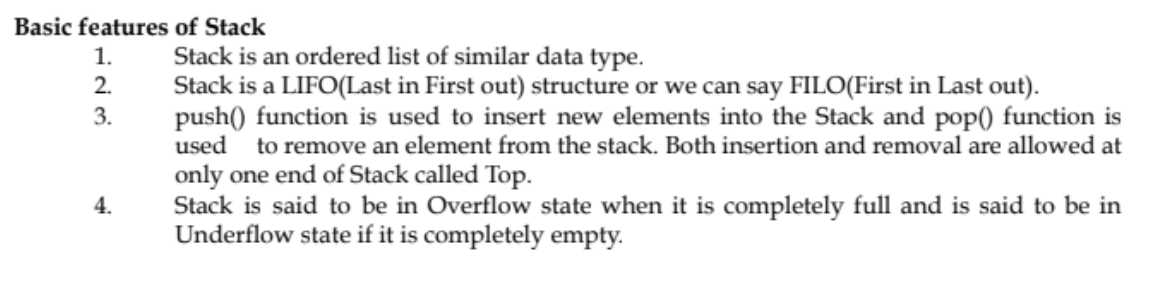
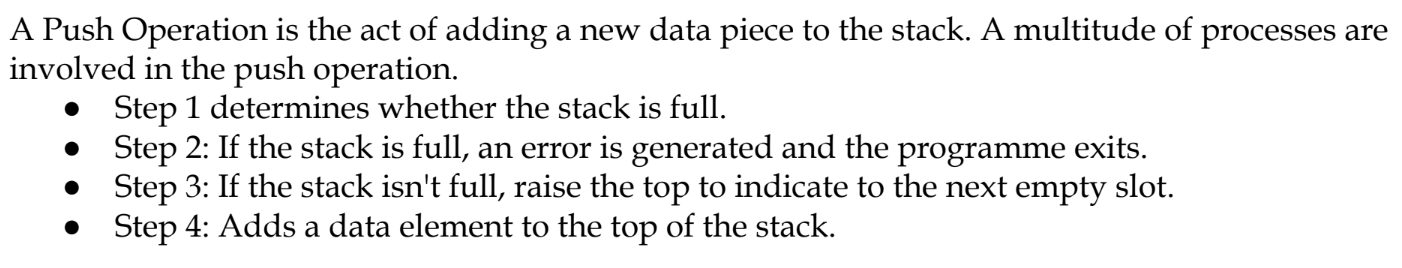
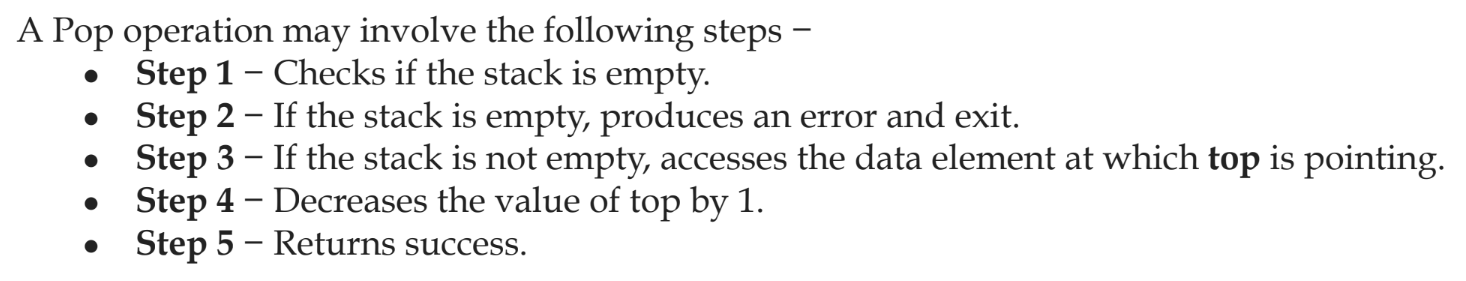
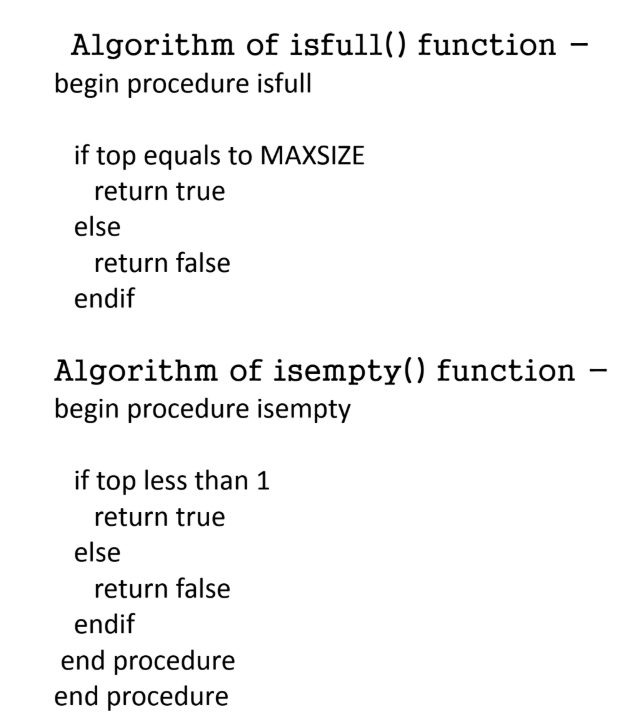
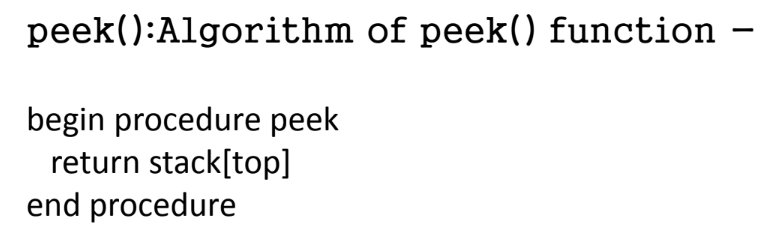
Chapter 1: ISE 1



Module 2:  








“**Stack is a collection of data, whose elements are added and removed at one end called top**.”

Algorithms:

**PUSH**

Step 1: If TOP >= SIZE – 1 then Write “Stack is Overflow”

Step 2: TOP = TOP + 1

Step 3: STACK [TOP] = X

**POP**

Step 1: If TOP == -1 then Write “Stack is Underflow”

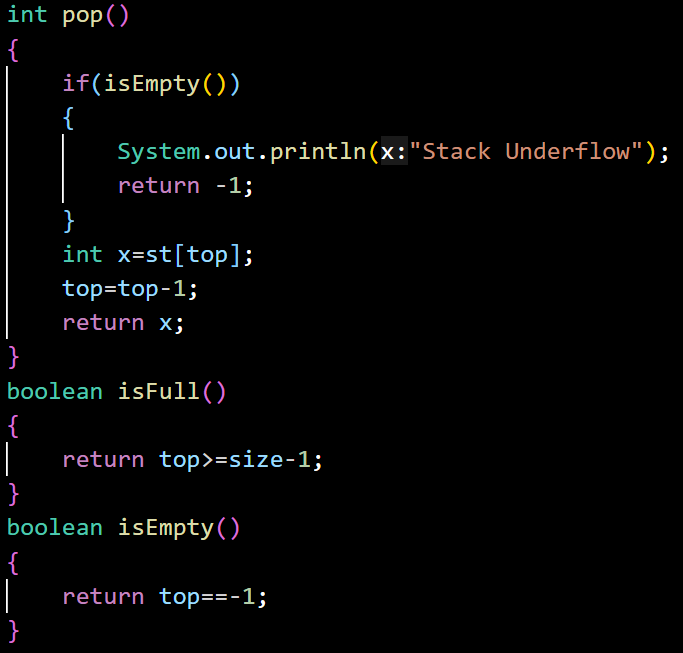
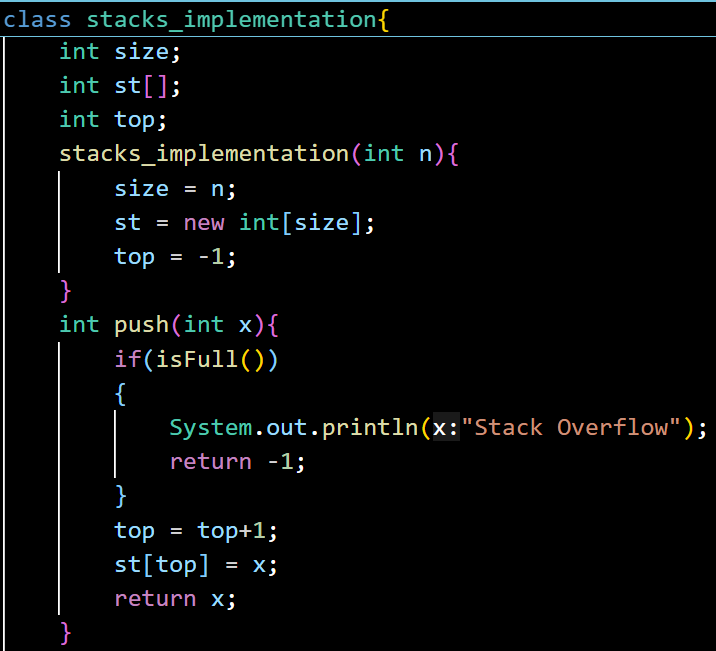
Step 2: Return STACK [TOP]

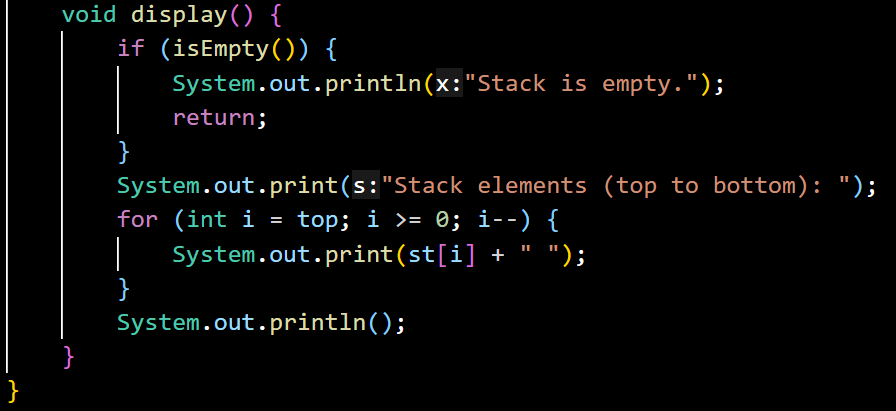
Step 3: TOP = TOP - 1

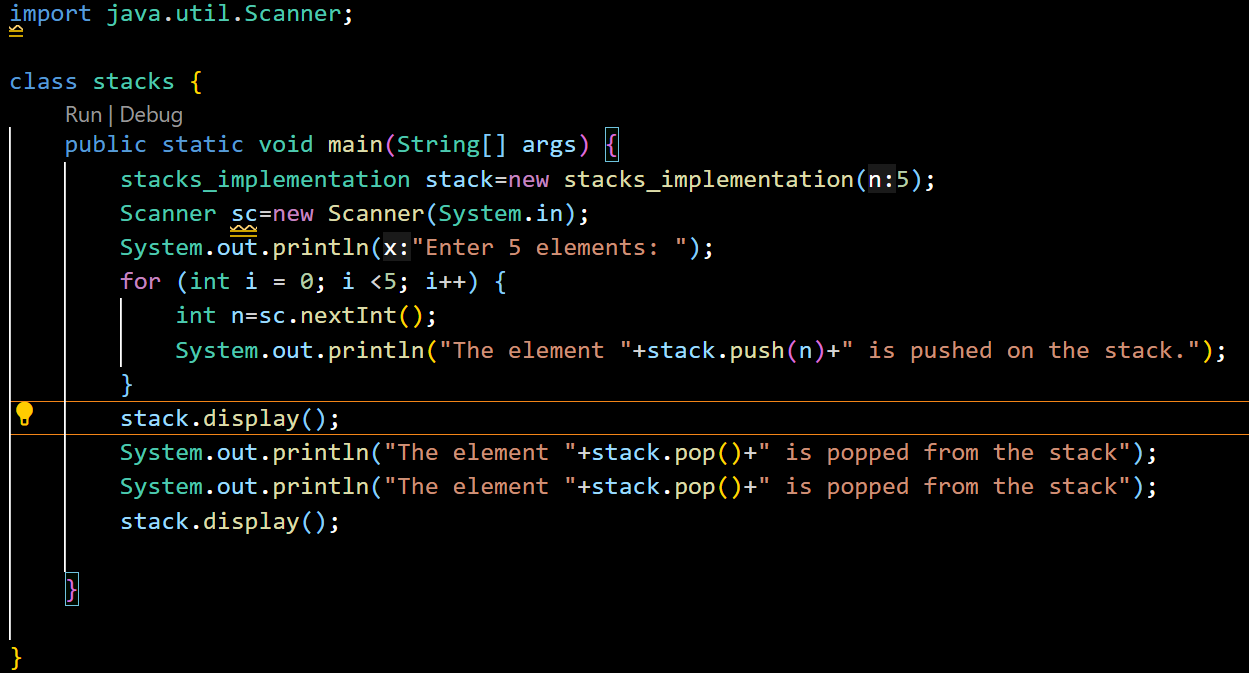
**PEEK**

Step 1: If TOP=-1 then Write “Stack is Underflow”

Step 2: Return STACK [TOP]







Applications of Stack:

1. **Recursion (using system stack)**
2. Every recursive function call is pushed onto the **call stack**.
3. When the function returns, its stack frame is popped out.
4. Example: Calculating factorial fact(5) → pushes calls fact(4), fact(3), etc., then pops them in reverse order.

**2. Evaluation of a Postfix Expression**

1. Postfix (Reverse Polish Notation) does not need parentheses.
2. Use a stack:
3. **Operands** → push to stack
4. **Operators** → pop two operands, apply operator, push result back
5. Example: 3 4 \* 2 + → push 3, 4, multiply → push 12, then add 2 → result 14.

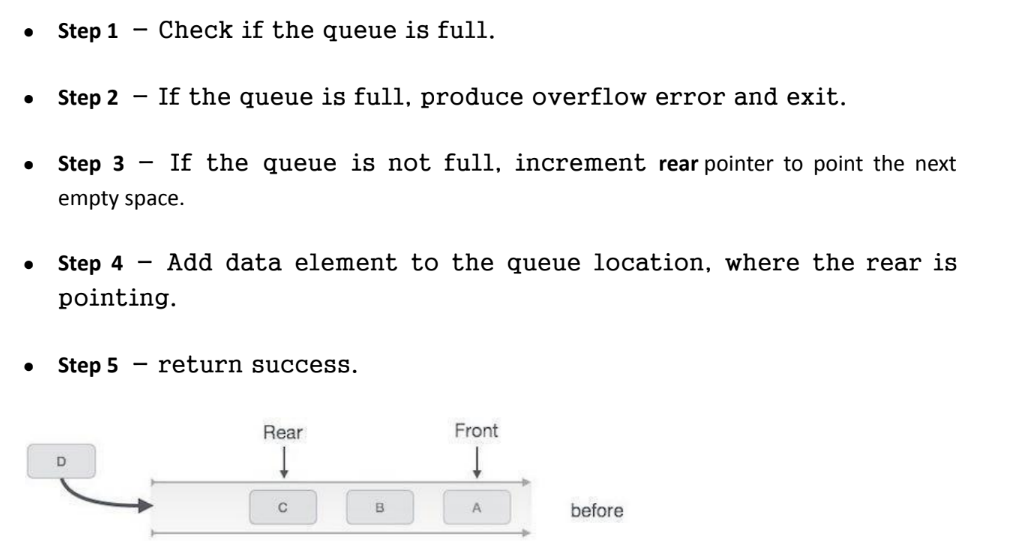
**3. Job Scheduling (CPU scheduling simulation)**

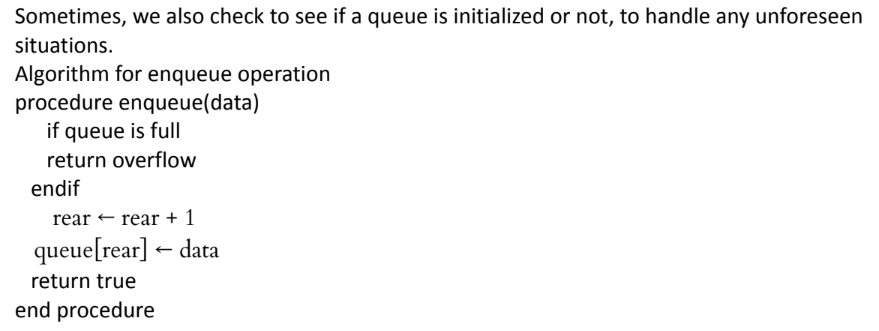
1. Stack can be used to manage jobs in **Last-In-First-Out (LIFO)** order.
2. Example: In **backtracking or undo operations**, the last job added is the first one executed or rolled back.
3. While real OS scheduling often uses queues, stack scheduling is useful for recursive or dependency-based jobs.

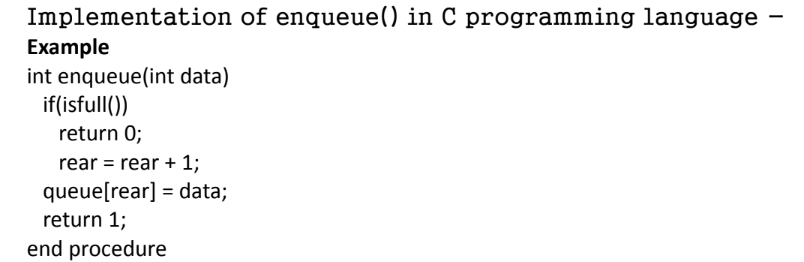
**4. Reverse a String**

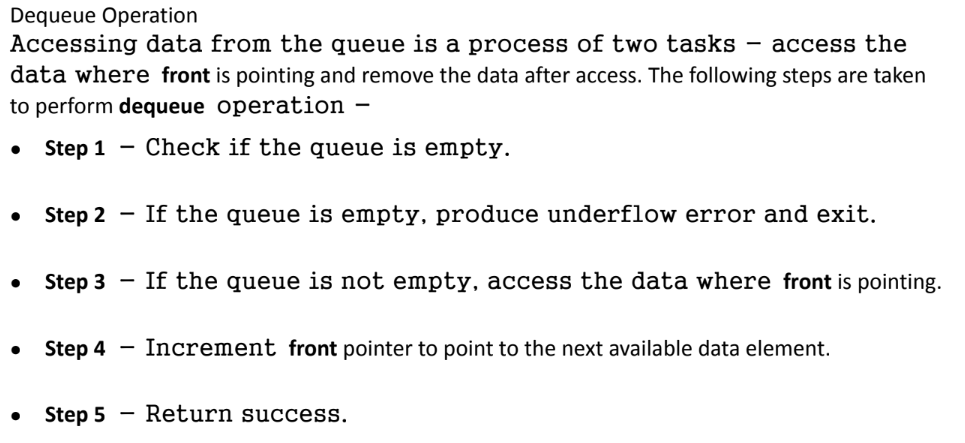
1. Push each character of the string into a stack.
2. Pop characters one by one → they come out in reverse order.
3. Example: "STACK" → push(S,T,A,C,K) → pop gives "KCAT S".

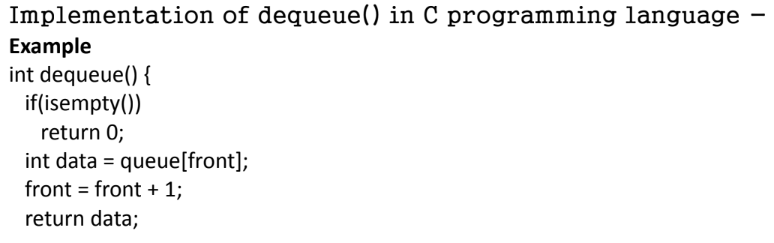
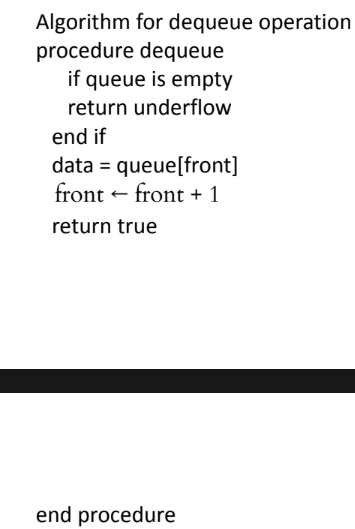
Queues:



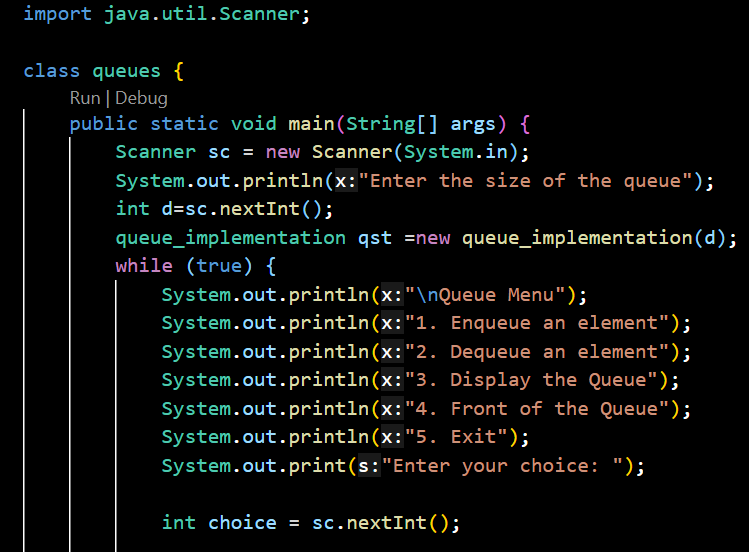


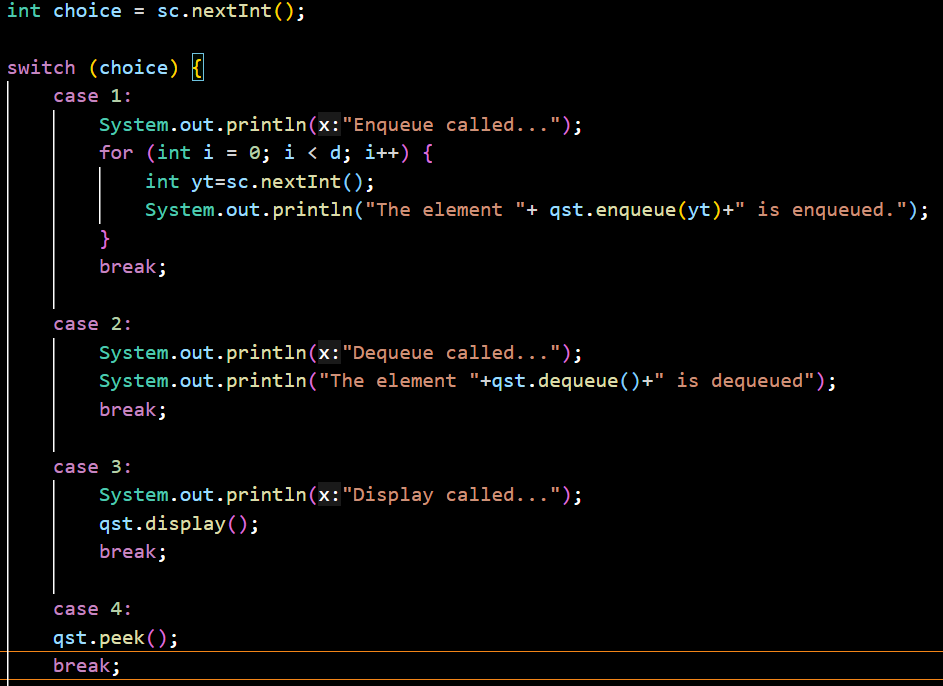


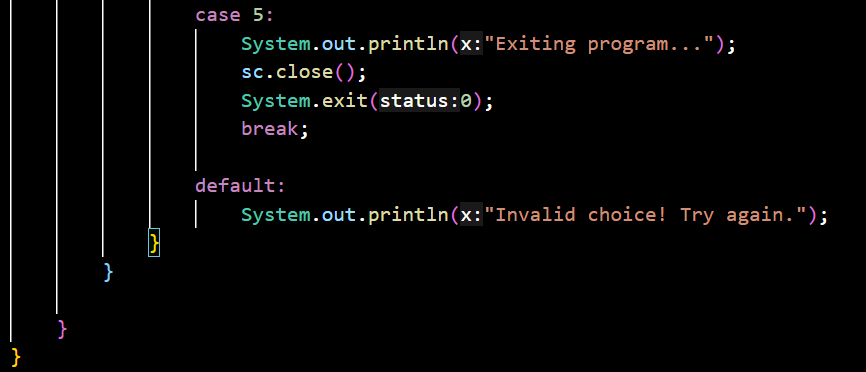


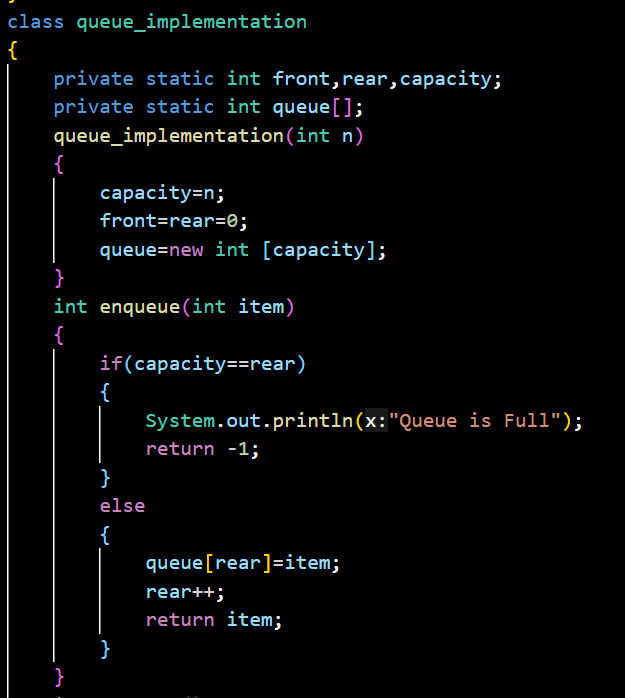


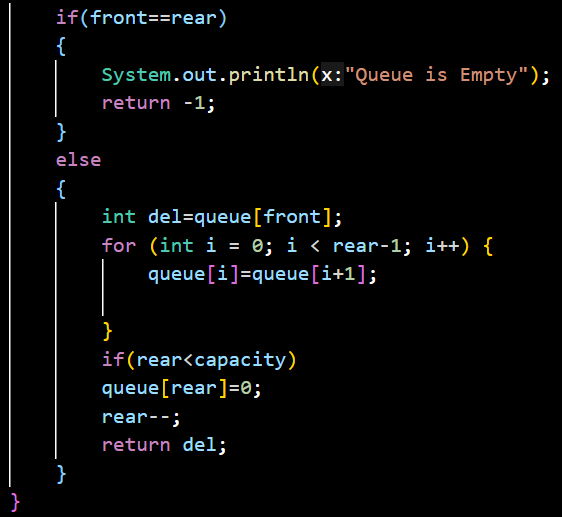
Program:

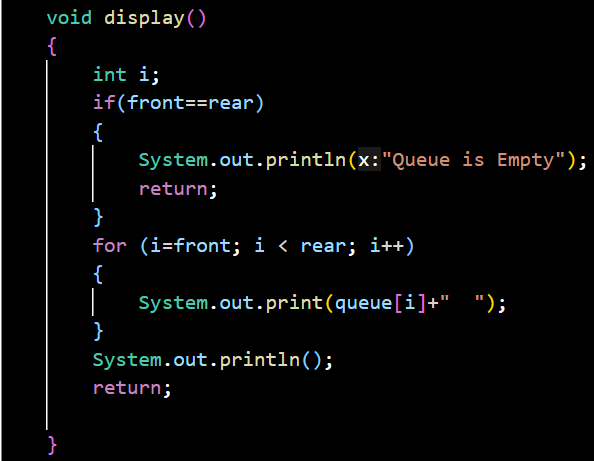


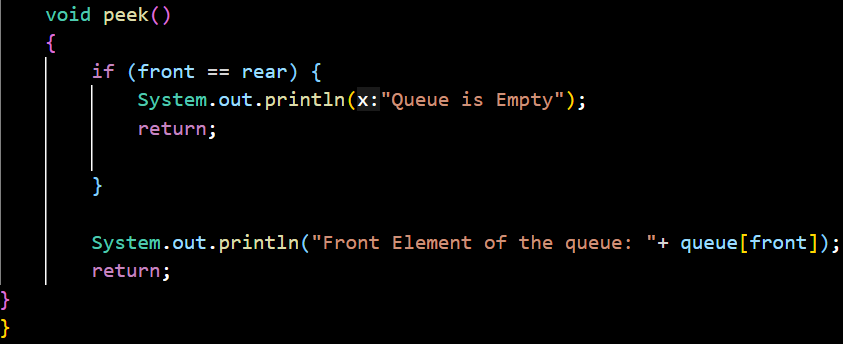




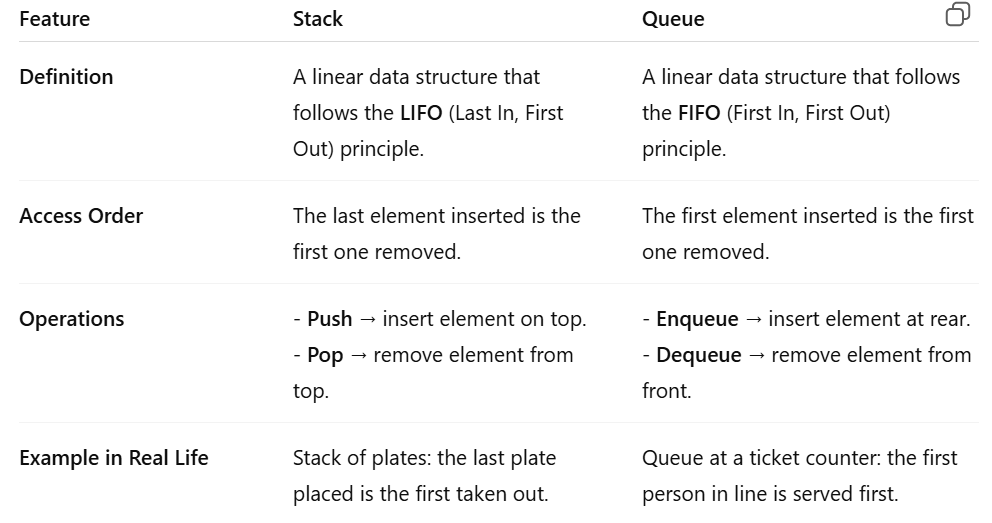


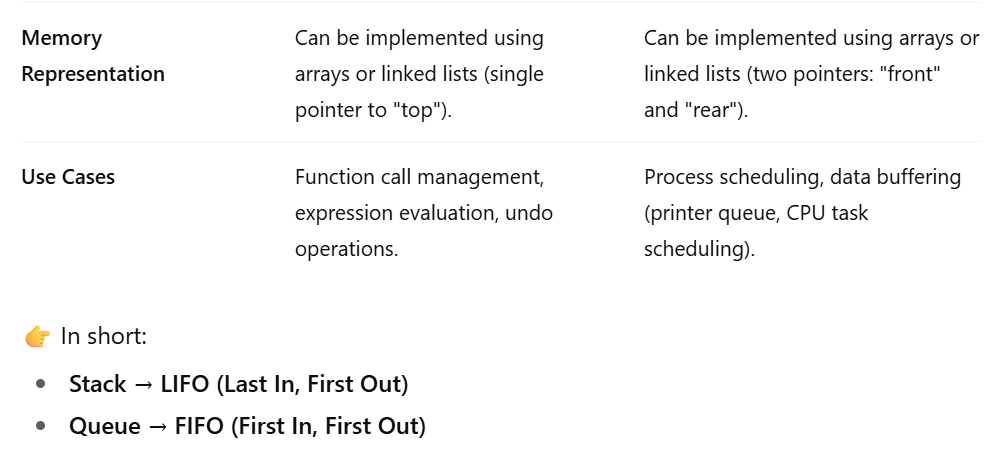






Queues and Stacks Difference:





Circular Queue

import java.util.Scanner;

class cirque {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        circularQueue cir = new circularQueue(5);

        cir.insert(5);

        cir.insert(4);

        cir.insert(3);

        cir.insert(2);

        cir.insert(1);

        cir.display();

        cir.delete();

        cir.display();

        cir.insert(65);

        cir.display();

        cir.insert(87);

    }

}

class circularQueue {

    private int front, rear, capacity;

    private int CQ[];

    circularQueue(int n) {

        capacity = n;

        front = rear = -1;

        CQ = new int[capacity];

    }

    boolean isFull() {

        return (rear + 1) % capacity == front;

    }

    boolean isEmpty() {

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

    }

    void insert(int item) {

        if (isFull()) {

            System.out.println("Queue is full. Cannot insert " + item);

            return;

        }

        if (isEmpty()) {

            front = 0;

            rear = 0;

        } else {

            rear = (rear + 1) % capacity;

        }

        CQ[rear] = item;

        System.out.println(item + " is inserted.");

    }

    void delete() {

        if (isEmpty()) {

            System.out.println("Queue is Empty.");

            return;

        }

        System.out.println(CQ[front] + " is deleted.");

        if (front == rear) {

            front = -1;

            rear = -1;

        } else {

            front = (front + 1) % capacity;

        }

    }

    void display() {

        if (isEmpty()) {

            System.out.println("Queue is Empty. No elements to display");

            return;

        }

        System.out.print("Queue elements: ");

        int i = front;

        while (true) {

            System.out.print(CQ[i] + " ");

            if (i == rear) {

                break;

            }

            i = (i + 1) % capacity;

        }

        System.out.println();

    }

}