Lahore Garrison University

Assignment # 1

Topic: **Stack Conversion & Double Ended Queue (Deque)**

Instructor: Ms. Farwa Javed	Semester: Fall 2024 (3rd)
Submission Deadline: 29-OCT-2025	Section:
Course: CS-6313- Data Structures	Date: 20-OCT-2025
Class: BS Computer Science	Name:
Marks: 20	Roll Number:

General Guidelines

1. This assignment consists of **three parts (A, B, and C)**.
   Each part assesses your ability to apply data structure concepts (Deque, Queue, and Stack Conversions) to practical and theoretical problems.
2. The assignment is based on **CLO-3:** *Apply the knowledge of data structure to other application domains (C-3)*.

Part A & Part B — System Implementation (Practical Work)

1. **Part A (Double-Ended Queue)** and **Part B (Restricted Queue)** must be implemented and executed **on your computer system** using **C++ language**.
2. You are required to:
   - Write **well-commented source code** for each part.
   - Include **output screenshots** showing correct working (insertions, deletions, overflow, underflow, or mode restriction).
   - Answer the given **conceptual questions** at the end of each part.
3. Each program must include:
   - Proper **class structure** and **function definitions** as instructed.
   - **Error handling** for invalid operations.
   - A **menu-driven interface** or function testing sequence.
4. Save your work in a single report document with the following order:
   - Cover Page
   - Part A Code + Output + Answers
   - Part B Code + Output + Answers

Part C — Handwritten Solution (Theoretical Work)

1. **Part C (Stack Conversion)** must be solved **by hand** neatly written

2. You must fill the **dry-run tables** step by step (showing stack contents and expressions formed).

Submission Requirements

- Submit your final assignment as a **single PDF or Word document** containing:

Page1: Cover Page: Name, Roll No., Section, Course Title, Assignment No.
Page2: Introduction: Problem overview and purpose of deque.
Page3: Algorithm / Flowchart: Step-by-step design logic.
Page4: Code Implementation: Well-commented C++ source code.
Page5: Output Screenshots: Showing insertions, deletions, overflow, underflow.
Page6: Answers to Assignment Questions.

- **Deadline:** 29-OCT-2025
- Late submissions will result in **mark deductions**.

Important Notes

- Code copying or identical submissions will lead to **zero marks** for all involved students.
- Each student must understand their code and logic; **viva or oral questioning** may be conducted for verification.
- Maintain **academic integrity** and submit **original work** only.

PART A — Double-Ended Queue (Deque)

Scenario-Based Case Study: Parking Lot Management System

Modern smart cities use **automated parking systems** that manage vehicle entry and exit from multiple gates. To ensure smooth traffic flow, vehicles can enter or leave from **either side of the parking area** depending on congestion.
In this project, you will simulate such a system using a **Double-Ended Queue (Deque)**, a linear data structure that allows **insertion and deletion at both the front and the rear**. This flexibility makes it ideal for real-world systems like parking management, traffic lanes, and scheduling queues.
You are a **computer science student** working with a **smart parking company** that manages a **small, automated parking lot**.
Cars can **enter or exit** from **either gate** — the **North Gate** or the **South Gate** — depending on real-time conditions. Your task is to **design and implement a Deque-based parking management system** that efficiently manages the entry and exit of vehicles.

System Description

Operation	Description	Function to Use
Car enters from **North Gate**	Insert at the **front**	`insertFront(int x)`
Car entry from **South Gate**	Insert at the **rear**	`insertRear(int x)`
Car leaves from **North Gate**	Delete from the **front**	`deleteFront()`
Car leaves from **South Gate**	Delete from the **rear**	`deleteRear()`
Display cars in parking lot	Show current order of cars	`display()`

Hints for Implementation

1. Use either an **array-based circular deque** or a **linked list**.
2. Maintain two pointers:
   - `front` → index of first element
   - `rear` → index of last element
3. **Overflow condition** occurs when the parking lot is full.
4. **Underflow condition** occurs when the parking lot is empty.
5. Apply **wrap-around logic** in circular arrays.
6. Handle edge cases where only one car remains.

C++ Functions to Implement

1. `insertFront(int x)` → Car enters from North (front side).
2. `insertRear(int x)` → Car enters from South (rear side).
3. `deleteFront()` → Car exits from North.
4. `deleteRear()` → Car exits from South.
5. `display()` → Shows current parking state (front → rear).

Sample Operation Sequence

Operation	Description	Parking State (Front → Rear)
insertRear(101)	Car 101 enters from south	[101]
insertRear(102)	Car 102 enters from south	[101, 102]
insertFront(999)	VIP car 999 enters from north	[999, 101, 102]
deleteRear()	Car exits from south	[999, 101]
deleteFront()	Car exits from north	[101]

After implementing the code, answer the following questions:

1) Explain why a *Double-Ended Queue (Deque)* is more suitable for a parking system with two gates compared to a simple queue or stack.
2) Draw a diagram showing the **state of the parking lot (deque)** after each of the following operations:
   insertRear(10), insertRear(20), insertFront(99), deleteRear(), insertFront(55), deleteFront().
   Label the **front** and **rear pointers** clearly at every step.
3) Assume the parking lot has a capacity of 5 cars. What will happen if you perform insertRear(105) when all slots are full?
   Explain how your program detects and handles this condition.
   Also, write the if-condition in code that checks for overflow.
4) The company introduces a **VIP priority rule**:
   VIP cars always enter from the **north gate** and must **never be removed automatically** even if the lot is full.
   Suggest **one modification** to your program logic or algorithm to support this rule without breaking the existing deque structure.

PART B — Restricted Queue

Scenario-Based Case Study: Print Management System

You are assigned to develop a **print management system** for your university's computer lab.
Printers receive print requests from multiple users, but the administrator wants to **control** how jobs are added or removed from the print queue.
Your task is to design the system using **Restricted Deques**, where insertion or deletion is limited at one end.

System Description

There are two modes of restriction:

1. **Input-Restricted Queue:**
   - Insertion (new print jobs) is allowed **only at the rear**.
   - Deletion (cancel or complete jobs) is allowed from **both ends**.
   - Simulates a queue where only the printer adds tasks, but admin can cancel from front or back.
2. **Output-Restricted Queue:**
   - Deletion is allowed **only from the front** (jobs processed in order).
   - Insertion allowed at **both ends** (urgent jobs can be inserted at the front).
   - Simulates a system where **urgent print requests** can skip the line.

Hints for Implementation

- Use your deque structure from Part A.
- For **input-restricted mode**, disable insertFront().
- For **output-restricted mode**, disable deleteRear().
- Add a variable mode to toggle between queue types.

Example Operations

Input-Restricted Queue Example

Operation	Description	Queue State (Front → Rear)
insertRear(10)	Add new print job	[10]
insertRear(20)	Add next job	[10, 20]
deleteFront()	Printer finishes front job	[20]
deleteRear()	Admin cancels last job	[]

Output-Restricted Queue Example

Operation	Description	Queue State (Front → Rear)
insertRear(11)	Normal job added	[11]
insertFront(999)	Urgent job inserted	[999, 11]
deleteFront()	Process next job	[11]

Challenge Task

Modify your deque program to accept a mode selection:

- Mode 1 → Input Restricted
- Mode 2 → Output Restricted

Disable the appropriate operations dynamically.

After implementing the code, answer the following questions:

1. What is the **key difference** between an Input-Restricted and Output-Restricted Deque?
2. In the **print management system**, why might we restrict input or output?
3. What are the **advantages** of using a Deque-based approach over a simple queue?
4. Which real-world scenarios could use an Output-Restricted Deque?
5. How can restricting operations help improve **system control or security**?
6. Can an Input-Restricted Deque act as a normal queue? Why or why not?
7. Which implementation (linked list or array) would better handle **dynamic print job loads**, and why?

PART C- Stack Conversion

Q1: You are required to design and implement a **Prefix → Infix conversion program** using a **stack-based approach** that performs the following:

1. **Accepts a prefix expression** as input (e.g., `- + * A B / C D ^ E F`).
2. **Processes it from right to left**, using stack operations.
3. **Outputs the equivalent infix expression** with appropriate parentheses.
4. Displays the **step-by-step dry run** showing stack content and expression formed after each operation.
5. For example:

A compiler parses: `- + * A B / C D ^ E F`
It must display: `(((A * B) + (C / D)) - (E ^ F))`

Pseudocode

```
1. Initialize an empty stack.
2. Scan the prefix expression from RIGHT → LEFT:
   a. If the current symbol is an OPERAND:
        push(symbol)
   b. If the current symbol is an OPERATOR:
        op1 = pop()
        op2 = pop()
        expr = "(" + op1 + symbol + op2 + ")"
        push(expr)
3. The final item in the stack is the full infix expression.
```

Dry run Table:

Step	Current Symbol	Stack (Top→Bottom)	Action / Notes	Infix Expression Formed	Code Line Executed
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					

Q2: You are required to design and implement a **Prefix → Postfix conversion program** using a **stack-based algorithm** that performs the following:

1. **Accepts a prefix expression** as input (e.g., `- + * A B / C D ^ E F`).
2. **Processes the expression from right to left.**
3. **Use a stack** to temporarily store operands and operators.
4. **Generates the equivalent postfix expression** (suitable for direct evaluation by a stack machine).
5. Displays a **step-by-step dry run** of the conversion process showing how the stack changes at each step.
6. **Pseudocode:**

```
1. Initialize an empty stack.
2. Scan prefix expression from RIGHT → LEFT:
   a. If symbol is OPERAND:
        push(symbol)
   b. If symbol is OPERATOR:
        op1 = pop()
        op2 = pop()
        expr = op1 + op2 + symbol
        push(expr)
3. Final stack top = complete postfix expression.
```

Dry run Table:

Step	Symbol	Stack (Top→Bottom)	Action / Notes	Infix Formed	Code Line Executed
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					

Q3: An **AI-based symbolic engine** has generated the following expression in **postfix form** for internal computation:

A B C * D + E F G * + ^ H I / -

Your task is to design and implement a **Postfix → Infix Conversion algorithm** that can automatically reconstruct the equivalent **infix expression**, maintaining correct **operator precedence** and **parentheses placement**.

```
((A ^ ((B * C + D) + (E + (F * G)))) - (H / I))
```

Pseudocode:

```
1. Initialize empty stack.
2. For each symbol in postfix expression:
     a. IF symbol is OPERAND → PUSH(symbol)
     b. ELSE IF symbol is OPERATOR →
           op2 = POP()
           op1 = POP()
           expr = "(" + op1 + symbol + op2 + ")"
           PUSH(expr)
3. END FOR
4. Final stack top → Resulting infix expression.
```

Dry run table:

Step	Symbol	Stack (Top→Bottom)	Action / Notes	Infix Formed	Code Line Executed
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					

Q4: An AI symbolic reasoning engine (e.g., a theorem prover or symbolic transformer) uses **postfix** expressions for efficient evaluation. For some analyses — such as generating expression trees, transformation rules, or communicating internal reasoning — the engine needs to convert postfix expressions into **prefix (Polish)** notation, which maps naturally to tree representations.

Postfix:

A B C * D + E F G * + ^ H I / -

Expected Prefix:

- ^ A + + * B C D + E * F G / H

```
1. Create empty stack.
2. For each symbol in postfix (LEFT → RIGHT):
     a. IF symbol is OPERAND → PUSH(symbol)
     b. ELSE IF symbol is OPERATOR →
           op2 = POP()
           op1 = POP()
           expr = symbol + " " + op1 + " " + op2
           PUSH(expr)
3. END FOR
4. Return top of stack as final prefix expression.
```

Dry run table:

Step	Symbol	Stack (Top→Bottom)	Action / Notes	Infix Formed	Code Line Executed
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					