Data Structures and Algorithms Lab

Lab 05 **Marks 10**

Instructions

- Work in this lab individually. Follow the best coding practices and include comments to explain the logic where necessary.
- You can use your books, notes, handouts, etc. but you are not allowed to borrow anything from your peer student.
- Do not use any **AI** tool for help; doing so will be considered cheating and may result in lab cancellation and possible disciplinary action.
- Test your program thoroughly with various inputs to ensure proper functionality and error handling.
- Show your work to the instructor before leaving the lab to get some or full credit.

ADT: Stack

Implement the following generic **Stack** class to provide the standard stack structure of **LIFO (Last-In, First-Out)** as discussed in the class.

```
template <class T>
class Stack
{
public:
        // Constructor
        Stack(const int MAX_SIZE = 10);  // Default MAX_SIZE set to 10

        // Destructor
        ~Stack();

        // Stack manipulation operations
        void push(const T newItem);      // Push a new item onto the stack
        void pop();                      // Pop an item from the stack
        void clear();                    // Clear the stack

        // Stack accessor
        T getTop() const;                // Return item at the top of the stack

        // Stack status operations
        bool isEmpty() const;            // Check if the stack is empty
        bool isFull() const;             // Check if the stack is full

        // Outputs the data in the stack.
        // If the stack is empty, output "Empty Stack".
        // If not, display elements from top to bottom.
        void showStructure() const;

private:
        // Data members
        T* data;                  // Array of items (dynamically allocated based no MAX_SIZE)
        int top;                  // Index of the top item in the stack
        const int MAX_SIZE;       // Maximum capacity of the stack
};
```

Note:

Ensure that you **handle errors gracefully**, **throw an exception**, or provide **meaningful error messages** when necessary (e.g., when attempting to **push** to a **full stack** or **pop** from an **empty stack**). **Pay attention to memory management** to avoid leaks and ensure proper resource cleanup.

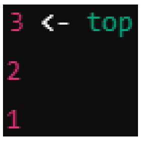
Demonstration:

In the **main** function:

1. **Create objects** of the **Stack** class for various data types (e.g., `int`, `float`, `string`).
2. **Test all implemented functions** (`push`, `pop`, `clear`, `getTop`, `isEmpty`, `isFull`, and `showStructure`).
3. **Ensure the stack operates correctly** under various conditions, including **edge cases**.

Stack Display (`showStructure`):

When displaying the stack's contents using `showStructure`, indicate the **top** of the stack. For example, if the stack contains [1, 2, 3], where **3 is the top**, it should be displayed as:



```
3 <- top
2
1
```

ADT: Queue

Implement the following generic **Queue** class to provide the standard **circular queue** structure using **FIFO (First-In, First-Out)**, as discussed in class.

```
template <class T>
class Queue
{
public:
        // Constructor
        Queue(const int MAX_SIZE = 5);   // Default MAX_SIZE set to 10

        // Destructor
        ~Queue();

        // Queue manipulation operations
        void enQueue(const T newItem);   // Enqueue a new item onto the queue
        void deQueue();                  // Dequeue an item from the queue
        void clear();                    // clear the queue

        // Queue accessors
        T getFront() const;              // Return item at the front of the queue
        T getRear() const;               // Return item at the rear of the queue

        // Queue status operations
        bool isEmpty() const;            // Check if the queue is empty
        bool isFull() const;             // Check if the queue is full

        // Outputs the data in the queue.
        // If the queue is empty, output "Empty Queue".
        // If not, display elements from front to rear.
        void showStructure() const;

private:
        // Data members
        T* data;                  // Array of items (dynamically allocated based no MAX_SIZE)
        int front;                // Index of the front item in the queue
        int rear;                 // Index of the rear item in the queue
        const int MAX_SIZE;       // Maximum capacity of the queue
};
```

Note:

Ensure that you **handle errors gracefully**, **throw an exception**, or provide **meaningful error messages** when necessary (e.g., when attempting to **enqueue** to a **full queue** or **dequeue** from an **empty queue**). **Pay attention to memory management** to avoid leaks and ensure proper resource cleanup.

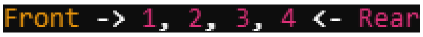
Demonstration:

In the **main** function:

1. **Create objects** of the **Queue** class for various data types (e.g., `int`, `float`, `string`).
2. **Test all implemented functions** (`enQueue`, `deQueue`, `clear`, `getFront`, `getRead`, `isEmpty`, `isFull`, and `showStructure`).
3. **Ensure the queue operates correctly** under various conditions, including **edge cases**.

Queue Display (`showStructure`):

When displaying the queue's contents using `showStructure`, indicate the **front** and **rear** of the queue. For example, if the queue contains $[1, 2, 3, 4]$, where **1 is the front** and **4 is the rear**, it should be displayed as:



```
Front -> 1, 2, 3, 4 <- Rear
```

Time Complexity

For both **Stack** and **Queue**:

- All operations should be performed in $\boldsymbol{O(1)}$ time complexity.
- The **showStructure** function may take $\boldsymbol{O(n)}$ time complexity, where ***n*** is the number of elements in the **stack** or **queue**.