# **Ch. 13 CSE3110 - Iterative Algorithm 1:** Error Log

Zephram Gilson

---

## StackList (+ StackListDemo)

One mistake I encountered was forgetting to update the top reference in the push method. Originally, I wrote:

```java
// Push operation to add an element to the top of the stack
public void push(Object data) {
    Node newNode = new Node(data);
    newNode.next = top;
}
```

The error was that the stack did not recognize the newly added element as the top, and operations like peek or pop still referenced the old top node. This caused the stack to behave incorrectly, with new elements being "lost."

```
<terminated> StackListDemo [Java Application] C:\Program Files\Eclipse\eclipse\plugins\org.eclipse.j
Pushing elements: 10, 20, 30
Exception in thread "main" java.lang.IllegalStateException: Stack is empty
        at mastery.StackList.StackList.peek(StackList.java:40)
        at mastery.StackList.StackListDemo.main(StackListDemo.java:14)
```

To fix this, I added the missing line:

```java
// Push operation to add an element to the top of the stack
public void push(Object data) {
    Node newNode = new Node(data);
    newNode.next = top;
    top = newNode;
}
```

This ensured that the new node was correctly assigned as the top of the stack, and the output was correct:

```
<terminated> StackListDemo [Java Application] C:\Progr
Pushing elements: 10, 20, 30
Top element (peek): 30

Popping elements:
30
20
10

Attempting to peek on an empty stack:
Caught exception: Stack is empty

Attempting to pop from an empty stack:
Caught exception: Stack is empty
```

## ReverseList

Initially, I thought I needed to push elements into another stack (or some other structure) to reverse their order before displaying them.

```
// Attempt to reverse the stack (incorrect logic)
System.out.println("\nReversed numbers:");
Stack<Integer> reversedStack = new Stack<>();  // New stack to store reversed elements
while (!stack.isEmpty()) {
    reversedStack.push(stack.pop());  // Push each element into the new stack
}

// Display reversed numbers by popping from the reversed stack
while (!reversedStack.isEmpty()) {
    System.out.print(reversedStack.pop() + " ");  // Pop and print each element
}

scanner.close();
```

This was unnecessary, as the `Stack` class in Java already maintains the correct order when elements are popped, meaning the `pop()` operation inherently returns the elements in reverse order of how they were added. I realized I can directly **pop** the elements from the original stack, which automatically gives me the elements in reverse order, and print them as I go.

```
public class ReverseList {
    public static void main(String[] args) {
        // Create a Scanner for user input and a Stack to store integers
        Scanner scanner = new Scanner(System.in);
        Stack<Integer> stack = new Stack<>();

        System.out.println("Enter up to 10 numbers to reverse, or enter 999 to finish early:");

        int count = 0;  // Counter to track the number of inputs
        while (count < 10) {  // Limit input to 10 numbers
            System.out.print("Enter number " + (count + 1) + ": ");
            int input = scanner.nextInt();

            if (input == 999) {  // Check if the user wants to stop early
                break;
            }

            stack.push(input);  // Push the valid input onto the stack
            count++;  // Increment the count of numbers entered
        }

        // Display reversed numbers by popping from the stack
        System.out.println("\nReversed numbers:");
        while (!stack.isEmpty()) {
            System.out.print(stack.pop() + " ");  // Pop each element to reverse order
        }
        scanner.close();
    }
}
```

This approach is more simple and leverages the natural properties of the stack to reverse the numbers with minimal effort and clearer logic.

## QueueList (+ QueueListDemo)

One error that I made was that I forgot to include the constructor that initializes the queue's state. Specifically, the constructor should initialize the `front` and `rear` pointers to `null` and the `size` to `0`.

```
package mastery.QueueList;

public class QueueList {
    private Node front;
    private Node rear;
    private int size;

    // Inner Node class
    private class Node {
        Object data;
        Node next;

        Node(Object data) {
            this.data = data;
            this.next = null;
        }
    }


    // Enqueue operation - Adds an item to the rear of the queue
    public void enqueue(Object item) {
        Node newNode = new Node(item);
        if (rear != null) {
            rear.next = newNode;
        }
        rear = newNode;
        if (front == null) {
            front = newNode;
        }
        size++;
    }
```

With the constructor in place, the `enqueue` and `dequeue` methods now behave correctly. The `enqueue` method properly links the new node to the `rear` pointer and updates the `front` pointer if the queue was empty. The `dequeue` method works because the `front` pointer is correctly initialized, allowing us to remove elements.

```
public class QueueList {
    private Node front;
    private Node rear;
    private int size;

    // Inner Node class
    private class Node {
        Object data;
        Node next;

        Node(Object data) {
            this.data = data;
            this.next = null;
        }
    }

    // Constructor
    public QueueList() {
        front = null;
        rear = null;
        size = 0;
    }

    // Enqueue operation - Adds an item to the rear of the queue
    public void enqueue(Object item) {
        Node newNode = new Node(item);
        if (rear != null) {
            rear.next = newNode;
        }
        rear = newNode;
        if (front == null) {
            front = newNode;
        }
        size++;
```

By adding the constructor, the queue is properly initialized, and the operations now work as intended.

These were the only errors I made throughout my coding process.