Report 4: Queue Implementation Using Double-Linked Linear List

Student Name: Jimmy 李皓鈞 Student ID: D1262028

1. Introduction

In this assignment, we were tasked with implementing a queue using a double-linked linear list in C++. This data structure is crucial for understanding fundamental operations such as enqueue and dequeue, which are widely applicable in software development, such as in operating systems and network traffic management.

2. Requirements Analysis

The core requirements of the project were to:

- Develop a queue that utilizes a double-linked linear list.

- Include operations to enqueue and dequeue elements.

- Ensure that the dequeue operation never attempts to remove more elements than the queue contains.

- Implement random testing for robustness.

3. Design

Node Class

The Node class is designed to store individual elements of the queue:

int elem: The element value.

Node* prev: Pointer to the previous node.

Node* next: Pointer to the next node.

Queue Class

The Queue class manages the queue operations:

Enqueue: Adds a new element to the end of the queue.

Dequeue: Removes an element from the front of the queue.

PrintHeadToTail: Displays all elements from front to back.

4. Implementation

The implementation involved defining the Node and Queue classes in C++.

Node Implementation

```
Node::Node(int val) : elem(val), prev(NULL), next(NULL) {}
```

Queue Implementation

Enqueue Method:

```
void IQueue::enqueue(int value) {
    Node* newNode = new Node(value);
    if (isEmpty()) {
        head = tail = newNode;
    } else {
        tail->next = newNode;
        newNode->prev = tail;
        tail = newNode;
```

```
        }
    }
```

Dequeue Method:

```
int IQueue::dequeue() {
        if (isEmpty()) return -1;
                int result = head->elem;
                Node* temp = head;
                head = head->next;
                delete temp;
    if (head == NULL) tail = NULL;
    else head->prev = NULL;
    return result;
  }
```

5. Challenges Encountered

Memory Management: Ensuring proper allocation and deallocation of memory to avoid leaks.

Edge Cases: Handling scenarios where dequeue was called on an empty queue.

6. Testing

The queue was tested with multiple scenarios:

Scenario 1: Enqueueing until full, then complete dequeue.

Scenario 2: Randomized enqueue and dequeue operations.

Results confirmed that the queue maintained integrity and order across all operations.

7. Lessons Learned

This project reinforced my understanding of dynamic data structures and pointer management in C++. It highlighted the importance of careful design to prevent common issues such as memory leaks and underflows.

8. Conclusion

The queue was successfully implemented and tested. Future improvements could include template implementation to support multiple data types and further optimization of memory usage.