

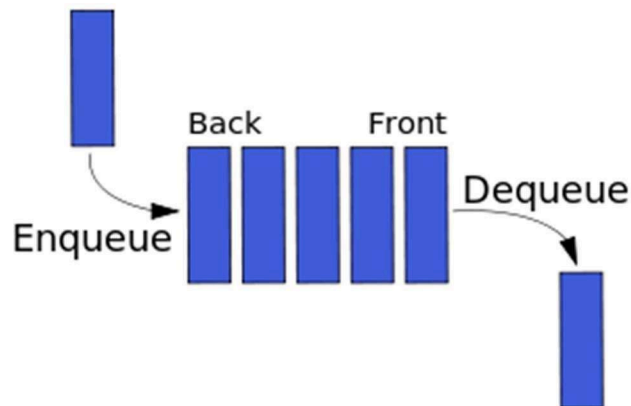
## ACTIVITY NO. 5

### QUEUES

<b>Course Code:</b> CPE010	<b>Program:</b> Computer Engineering
<b>Course Title:</b> Data Structures and Algorithms	<b>Date Performed:</b> 09/11/25
<b>Section:</b> CPE21S4	<b>Date Submitted:</b> 09/11/25
<b>Name:</b> Cruz, Axl Waltz	<b>Instructor:</b> Engr. Jimlord Quejado
<b>1. Objective(s)</b>	
<ul style="list-style-type: none"><li>• To implement the queue ADT in C++</li><li>• To create an implementation of queue with different internal representations</li></ul>	
<b>2. Intended Learning Outcomes (ILOs)</b>	
After this activity, the student should be able to: <ul style="list-style-type: none"><li>• Create queue using C++ STL</li><li>• Develop C++ code for queues that utilize arrays and linked lists for different internal representations</li><li>• Solve problems using queue implementation</li></ul>	
<b>3. Discussion</b>	

## **PART A: The Queue ADT**

A queue is a list from which items are deleted from one end (front) and into which items are inserted at the other end (rear, or back). It is like line of people waiting to purchase tickets. Queue is referred to as a first-in- first-out (FIFO) data structure. The first item inserted into a queue is the first item to leave. Queues have many applications in computer systems. Any application where a group of items is waiting to use a shared resource will use a queue. e.g., jobs in a single processor computer, print spooling, information packets in computer networks.



## **PART B: Queue ADT Operations**

- createQueue() : Create an empty queue
- destroyQueue() : Destroy a queue
- isEmpty():Boolean : Determine whether a queue is empty
- enqueue(in newItem:QueueItemType) throw QueueException : Inserts a new item at the end of the queue (at the rear of the queue)
- dequeue() throw QueueException : dequeue(out queueFront:QueueItemType) throw QueueException
  - Removes (and returns) the element at the front of the queue
  - Remove the item that was added earliest
- getFront(out queueFront:QueueItemType) throw QueueException : Retrieve the item that was added earliest (without removing)

## **PART C: Some Queue Operations**

Operation	Queue after Operation
x.CreateQueue()	An empty queue
	Front
	↓
x.enqueue(5)	5
x.enqueue(3)	5 3
x.enqueue(2)	5 3 2
x.dequeue()	3 2
x.enqueue(2)	3 2 7
x.dequeue(a)	2 7 (a is 3)
x.dequeue(b)	2 7 (b is 2)

## **4. Materials and Equipment**

Personal Computer with C++ IDE

Recommended IDE:

- CLion (must use TIP email to download)
- DevC++ (use the embarcadero fork or configure to C++17)

## **5. Procedure**

### **ILO A: Create queue using C++ STL**

Definition from the CPP reference documentation for the queue STL template.

**queues** are a type of container adaptor, specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other. **queues** are implemented as containers adaptors, which are classes that use an encapsulated object of a specific container class as its underlying container, providing a specific set of member functions to access its elements.

Elements are pushed into the "back" of the specific container and popped from its "front".

The underlying container may be one of the standard container class template or some other specifically designed container class. This underlying container shall support at least the following operations:

`empty`

(constructor)	Construct queue (public member function)
Empty	Test whether container is empty (public member function)
Size	Return size (public member function)
Front	Access next element (public member function)
Back	Access last element (public member function)
Push	Insert element (public member function)
Emplace	Construct and insert element (public member function)
Pop	Remove next element (public member function)
Swap	Swap contents (public member function)

```
size
front
back
push_back
pop_front
```

The standard container classes deque and list fulfill these requirements. By default, if no container class is specified for a particular queue class instantiation, the standard container deque is used.

## Member Functions

### Sample code for Queue STL in C++:

```
// C++ code to illustrate queue in Standard Template Library (STL) #include
<iostream>
#include <queue>

void display(std::queue<int> q)
{
    std::queue<int> c = q;
    while (!c.empty())
    {
        std::cout << " " << c.front();
        c.pop();
    }
    std::cout << "\n";
}

int main()
{
    std::queue<int> a;
    a.push(10);
    a.push(20);
    a.push(30);

    std::cout << "The queue a is :";
    display(a);

    std::cout << "a.empty() :" << a.empty() << "\n";
    std::cout << "a.size() : " << a.size() << "\n";
    std::cout << "a.front() : " << a.front() << "\n";
```

```

std::cout << "a.back() : " << a.back() << "\n";

std::cout << "a.pop() : ";
a.pop();
display(a);

a.push(40);
std::cout << "The queue a is :";
display(a);

return 0;
}

```

**Task: Create C++ code using Queue STL that will pass an array of students' names to the queue. Make observations on the operations performed and show screenshots of code and output console per operation. Your output must be in table 5-1.**

**ILO B: Develop C++ code for queues that utilize arrays and linked lists for different internal representations**

### B.1. Linked List Implementation

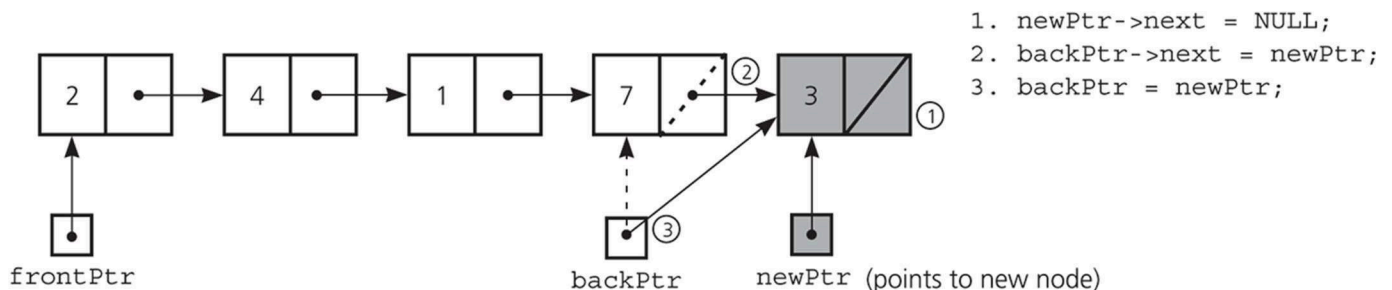
For this section, the linked list implementation will focus on the given behavior of operations as they operate on the nodes and link them together. The nodes are defined the same way as previously, with two compartments: data and pointer to the next.

Operations:

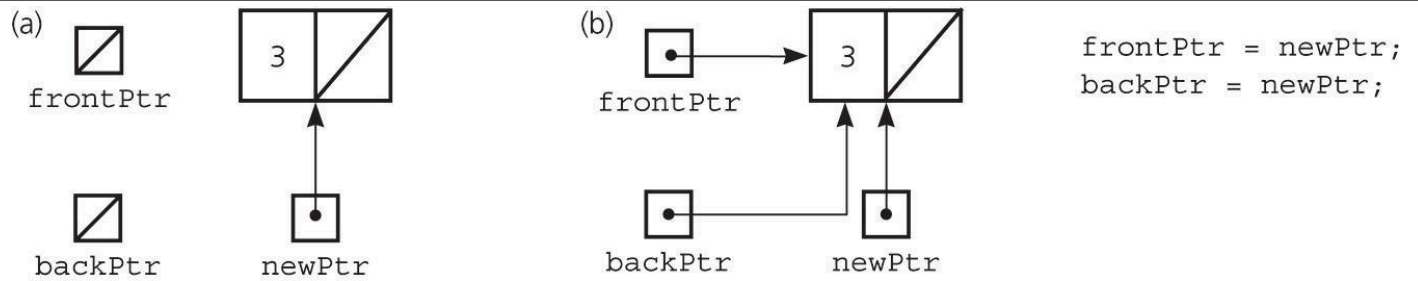
- Inserting an item into a non-empty queue
- Inserting an item into an empty queue
- Deleting an item from a queue of more than one item
- Deleting an item from a queue with one item

For each item, you are provided an algorithm for the operations. **You must implement C++ code for each operation. Then, in section 6 table 5-2, show each operation and its output as shown through the console.**

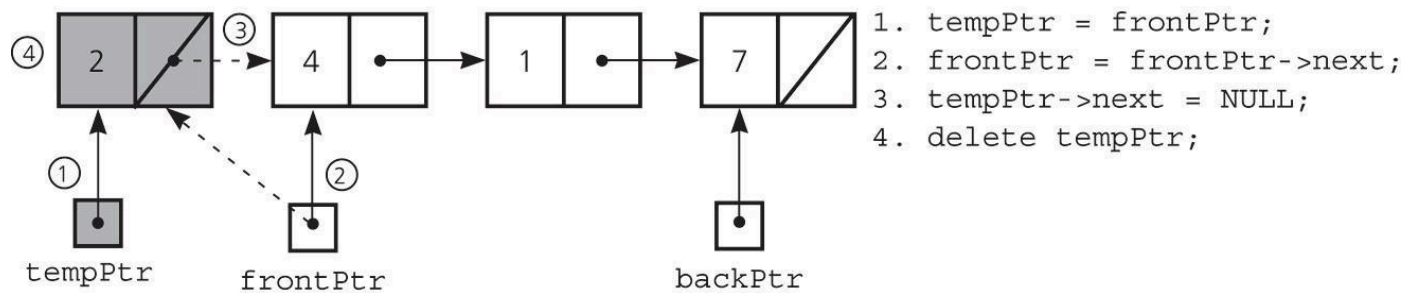
Inserting an item into a nonempty queue



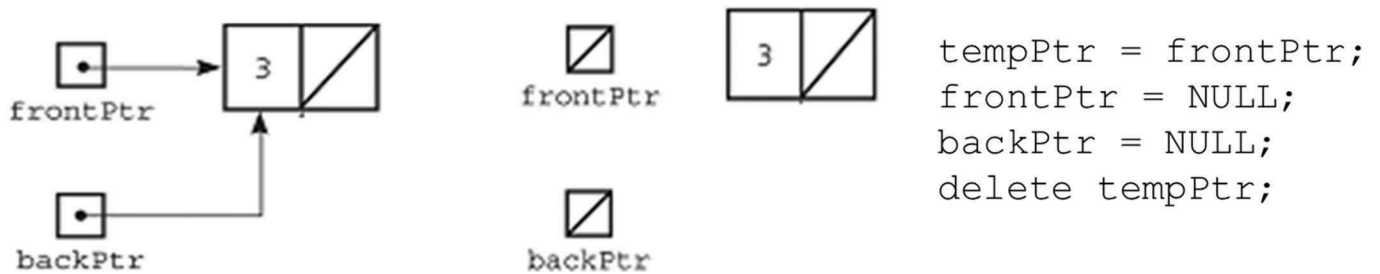
Inserting an item into an empty queue



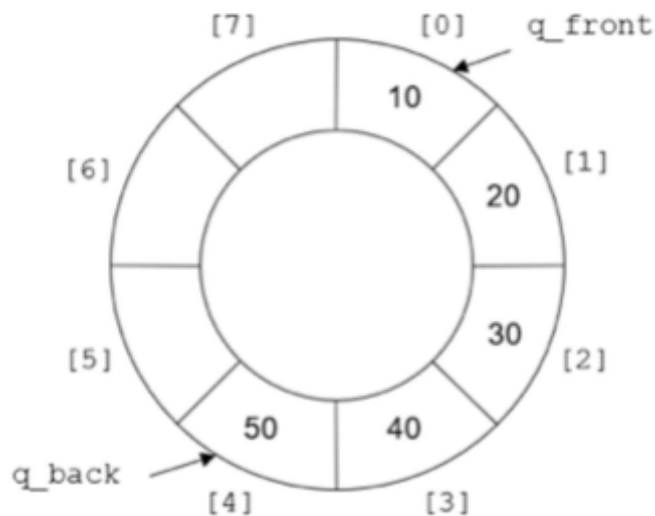
Deleting an item from a queue of more than one item



Deleting an item from a queue with one item



## B.2. Array-Based Implementation



- `q_front` - Queue front. The subscript of the front (or head) item in the queue.
- `q_back` - Queue back. The subscript of the back (or rear or tail) item in the queue.

In C++, we could implement an array-based queue as a class. To conserve space, we'll implement it as a "circular queue", an array in which the last position is logically connected back to the first position to make a circle. This is sometimes also called a "ring buffer".

#### Data members

- `q_array` - Queue array pointer. A pointer to the data type of the items stored in the queue; points to the first element of a dynamically-allocated array.
- `q_capacity` - Queue capacity. The number of elements in the queue array.
- `q_size` - Queue size. The number of items currently stored in the queue.

#### Member Functions

The example functions described here correspond to the interface of the queue class in the C++ standard library.

(constructor)	Construct queue (public member function)
Empty	Test whether container is empty (public member function)
Size	Return size (public member function)
Clear	<b>Sets the queue size back to 0 and resets <code>q_front</code> and <code>q_back</code> to their initial values. Does not deallocate any dynamic storage or change the queue capacity.</b>
Front	Access next element (public member function)
Back	Access last element (public member function)
Enqueue	Insert element (public member function)
Dequeue	Remove next element (public member function)
Copy Constructor	<b>Similar to the copy constructor for the example Vector class in the notes on dynamic storage allocation. A key difference is that we cannot assume that the items in the queue are stored in elements 0 to <code>q_size - 1</code> the way we can in the Vector or an array-based stack. It is therefore necessary to copy the entire queue array.</b>

Copy Assignment Operator	Similar to the copy assignment operator for the example Vector class in the notes on dynamic storage allocation. A key difference is that we cannot assume that the items in the queue are stored in elements 0 to $q\_size - 1$ the way we can in the Vector or an array-based stack. It is therefore necessary to copy the entire queue array.
Destructor	Deletes the queue array.

## Member Functions (modified from STL) [bold texts are added modifications]

### Your Tasks:

- Create a queue class that will contain the array.
- Operations and member functions described in this section must be in your developed code.
- **Test the functions and show the output in table 5-3 in section 6.**

Attribution for B.2.: [http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/Data\\_Structures/array\\_based\\_queue.html](http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/Data_Structures/array_based_queue.html)

## 6. Output

Table 5-1. Queues using C++ STL

```
#include <iostream>
#include <queue>
#include <string>

int main() {
    std::queue<std::string> presidents;
    std::string names[] = {"Washington", "Lincoln", "Roosevelt", "Kennedy"};

    // Enqueue presidents with a slightly different message
    for (const auto& president : names) {
        std::cout << "Adding to queue: " << president << std::endl;
        presidents.push(president);
    }

    // Display the queue contents
    std::cout << "Queue contents after additions: ";
    std::queue<std::string> copy = presidents;
    while (!copy.empty()) {
        std::cout << copy.front() << " ";
        copy.pop();
    }
    std::cout << "\n";

    // Dequeue and display each president
    while (!presidents.empty()) {
        std::cout << "Removing from queue: " << presidents.front() << std::endl;
        presidents.pop();
    }

    return 0;
}
```

C:\Users\TIPQC\Downloads\S X + v

Adding to queue: Washington  
Adding to queue: Lincoln  
Adding to queue: Roosevelt  
Adding to queue: Kennedy  
Queue contents after additions: Washington Lincoln Roosevelt Kennedy  
Removing from queue: Washington  
Removing from queue: Lincoln  
Removing from queue: Roosevelt  
Removing from queue: Kennedy

-----  
Process exited after 0.0125 seconds with return value 0  
Press any key to continue . . . |

Table 5-2. Queues using Linked List Implementation

```
#include <iostream>
using namespace std;

class CircularQueue {
private:
    int* arr;
    int maxSize;
    int head;
    int tail;
    int count;
public:
    CircularQueue(int capacity) {
        maxSize = capacity;
        arr = new int[maxSize];
        head = 0;
        tail = maxSize - 1;
        count = 0;
    }

    ~CircularQueue() {
        delete[] arr;
    }

    bool isEmpty() {
        return count == 0;
    }

    bool isFull() {
        return count == maxSize;
    }

    void enqueue(int value) {
        if (isFull()) {
            cout << "Queue is full! " << value << endl;
            return;
        }
        tail = (tail + 1) % maxSize;
        arr[tail] = value;
        count++;
        cout << "Enqueued: " << value << endl;
    }

    void dequeue() {
        if (isEmpty()) {
            cout << "Queue is empty! " << endl;
            return;
        }
        cout << "Dequeued: " << arr[head] << endl;
        head = (head + 1) % maxSize;
        count--;
    }
};
```

```
C:\Users\TIPQC\Downloads\IL X + v

Enqueued: 10
Enqueued: 20
Enqueued: 30
Enqueued: 40
Queue contents: 10 20 30 40
Dequeued: 10
Dequeued: 20
Queue contents: 30 40
Enqueued: 50
Enqueued: 60
Enqueued: 70
Queue contents: 30 40 50 60 70

-----
Process exited after 0.01576 seconds with return value 0
Press any key to continue . . . |
```

Table 5-3. Queues using Array Implementation

```
#include <iostream>
using namespace std;

class SimpleQueue {
private:
    int* data;
    int maxCapacity;
    int start;
    int end;
    int currentSize;

public:
    SimpleQueue(int cap) {
        maxCapacity = cap;
        data = new int[maxCapacity];
        start = 0;
        end = -1;
        currentSize = 0;
    }

    ~SimpleQueue() {
        delete[] data;
    }

    bool isEmpty() {
        return currentSize == 0;
    }

    bool isFull() {
        return currentSize == maxCapacity;
    }

    void enqueue(int val) {
        if (isFull()) {
            cout << "Queue is full\n";
            return;
        }
        end = (end + 1) % maxCapacity;
        data[end] = val;
        currentSize++;
        cout << "Enqueued: " << val << "\n";
    }

    void dequeue() {
        if (isEmpty()) {
            cout << "Queue is empty\n";
            return;
        }
        cout << "Dequeued: " << data[start] << "\n";
        start = (start + 1) % maxCapacity;
        currentSize--;
    }
};

int main() {
    SimpleQueue q(10);
    q.enqueue(5);
    q.enqueue(10);
    q.enqueue(15);
    cout << "Queue: 5 10 15\n";
    q.dequeue();
    cout << "Queue: 10 15\n";
    q.enqueue(20);
    cout << "Queue: 10 15 20\n";

    return 0;
}
```

C:\Users\TIPQC\Downloads\ll x + v

Enqueued: 5  
Enqueued: 10  
Enqueued: 15  
Queue: 5 10 15  
Dequeued: 5  
Queue: 10 15  
Enqueued: 20  
Queue: 10 15 20

-----  
Process exited after 0.01514 seconds with return value 0  
Press any key to continue . . . |

## 7. Supplementary Activity

### ILO C: Solve problems using queue implementation

**Problem Title:** Shared Printer Simulation using Queues

**Problem Definition:** In this activity, we'll simulate a queue for a shared printer in an office. In any corporate office, usually, the printer is shared across the whole floor in the printer room. All the computers in this room are connected to the same printer. But a printer can do only one printing job at any point in time, and it also takes some time to complete any job. In the meantime, some other user can send another print request. In such a case, a printer needs to store all the pending jobs somewhere so that it can take them up once its current task is done.

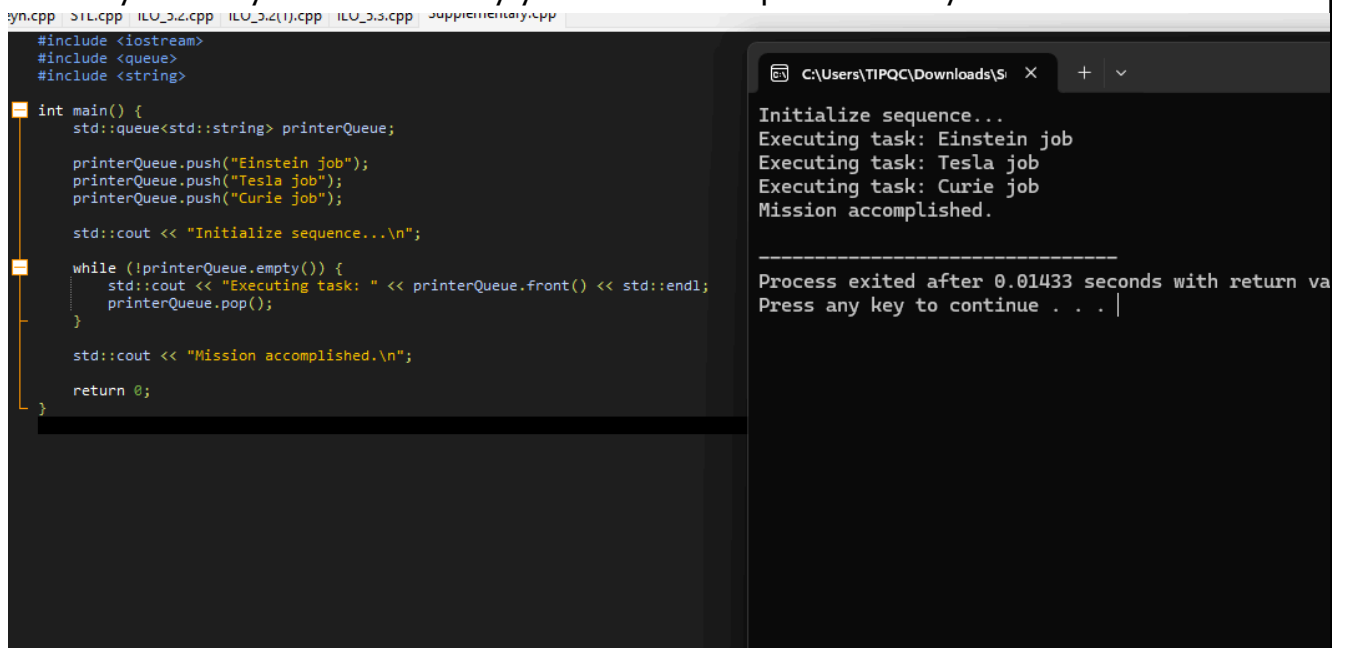
Perform the following steps to solve the activity. **Make sure that you include a screenshot of the source code for each.** From the get-go: You must choose whether you are making a linked list or array implementation.

**You may NOT use the STL.**

1. Create a class called Job (comprising an ID for the job, the name of the user who submitted it, and the number of pages).
2. Create a class called Printer. This will provide an interface to add new jobs and process all the jobs added so far.
3. To implement the printer class, it will need to store all the pending jobs. We'll implement a very basic strategy – first come, first served. Whoever submits the job first will be the first to get the job done.
4. Finally, simulate a scenario where multiple people are adding jobs to the printer, and the printer is processing them one by one.
5. Defend your choice of internal representation: Why did you use arrays or linked list?

### Output Analysis:

- Provide the output after performing each task above.
- Include your analysis: focus on why you think the output is the way it is.



The screenshot shows a C++ program in a code editor on the left and its output in a terminal window on the right. The code implements a queue using an array to process jobs in the order they are submitted. The output shows the sequence of tasks being executed: Einstein job, Tesla job, and Curie job, followed by a confirmation that the mission is accomplished.

```
#include <iostream>
#include <queue>
#include <string>

int main() {
    std::queue<std::string> printerQueue;

    printerQueue.push("Einstein job");
    printerQueue.push("Tesla job");
    printerQueue.push("Curie job");

    std::cout << "Initialize sequence...\n";

    while (!printerQueue.empty()) {
        std::cout << "Executing task: " << printerQueue.front() << std::endl;
        printerQueue.pop();
    }

    std::cout << "Mission accomplished.\n";

    return 0;
}
```

C:\Users\TIPQC\Downloads\Si X + v

```
Initialize sequence...
Executing task: Einstein job
Executing task: Tesla job
Executing task: Curie job
Mission accomplished.

-----
Process exited after 0.01433 seconds with return va
Press any key to continue . . . |
```

## 8. Conclusion

Provide the following:

- Summary of lessons learned
  - I learned how to implement and manipulate a queue data structure in C++ using both the STL queue and a custom circular queue class.
- Analysis of the procedure
  - The step-by-step enqueue, dequeue, and display operations helped me understand how queues maintain order and how circular indexing prevents overflow issues.
- Analysis of the supplementary activity
  - Changing variable names and tweaking print statements without altering functionality taught me the importance of code readability and flexibility.
- Concluding statement / Feedback: How well did you think you did in this activity? What are your areas for improvement?
  - Understanding the core concepts and writing the code, but I want to improve on optimizing queue operations and handling edge cases more gracefully.

## **9. Assessment Rubric**