Hands-On Activity 5.1	
Queues	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 10/7/24**
Section: CPE21S4	**Date Submitted: 10/7/24**
Name(s): Aries Rio	**Instructor: Ma'am Maria Rizette Sayo**

6. Output

Table 5.1

```cpp
// C++ code to illustrate queue in Standard Template Library (STL)
#include <iostream>
#include <queue>
#include <string>
using std::string;

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

int main()
{
    std::queue<string> a;
    a.push("Mark");
    a.push("Aries");
    a.push("Rio");

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

    return 0;
}
```

STDIN

Input for the program (Optional)

Output:

```
The queue a is : Mark Aries Rio
a.empty() : 0
a.size() : 3
a.front() : Mark
a.back() : Rio
a.pop() :  Aries Rio
The queue a is : Aries Rio Pogi
```

The C++ Queue STL works by adding elements to the back with push() and removing them from the front with pop(), following a First In, First Out (FIFO) order. You can check the first element using front() and the last one using back(). The size() function shows how many elements are in the queue, and empty() tells if the queue is empty. Each time you pop(), the oldest element is removed, and the rest shift forward.

Table 5.2

Main.cpp

42ucxa548

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

// Node structure
struct Node {
    int data;
    Node* next;
};

// Queue class using linked list
class Queue {
private:
    Node* front;
    Node* rear;

public:
    // Constructor to initialize an empty queue
    Queue() {
        front = rear = nullptr;
    }

    // Insert an item into the queue
    void enqueue(int value) {
        Node* newNode = new Node();
        newNode->data = value;
        newNode->next = nullptr;

        if (rear == nullptr) {  // If queue is empty
            front = rear = newNode;
            cout << "Inserted " << value << " into an empty queue.\n";
        } else {  // If queue has items
            rear->next = newNode;
            rear = newNode;
            cout << "Inserted " << value << " into a non-empty queue.\n";
        }
    }

    // Delete an item from the queue
    void dequeue() {
        if (front == nullptr) {  // If queue is empty
            cout << "Queue is empty, nothing to delete.\n";
            return;
        }

        Node* temp = front;
        front = front->next;

        if (front == nullptr) {  // If queue had only one item
            rear = nullptr;
            cout << "Deleted the last item from the queue.\n";
        } else {
            cout << "Deleted an item from a non-empty queue.\n";
        }

        delete temp;
    }

    // Display the queue contents
    void display() {
        if (front == nullptr) {
            cout << "Queue is empty.\n";
            return;
        }

        Node* temp = front;
        cout << "Queue: ";
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};

int main() {
    Queue q;

    q.enqueue(10);  // Inserting into an empty queue
    q.display();

    q.enqueue(20);  // Inserting into a non-empty queue
    q.enqueue(30);
    q.display();

    q.dequeue();  // Deleting from a queue with more than one item
    q.display();

    q.dequeue();  // Deleting the last item from the queue
    q.dequeue();
    q.display();

    return 0;
}
```

STDIN

Input for the program (Optional)

Output:

```
Inserted 10 into an empty queue.
Queue: 10
Inserted 20 into a non-empty queue.
Inserted 30 into a non-empty queue.
Queue: 10 20 30
Deleted an item from a non-empty queue.
Queue: 20 30
Deleted an item from a non-empty queue.
Deleted the last item from the queue.
Queue is empty.
```

Table 5.3

Main.cpp

42ucxa548

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

class CircularQueue {
private:
    int* q_array;    // Pointer to the queue array
    int q_capacity;  // Capacity of the queue
    int q_size;      // Current size of the queue
    int q_front;     // Front index
    int q_back;      // Back index

public:
    // Constructor to initialize the queue
    CircularQueue(int capacity) {
        q_capacity = capacity;
        q_array = new int[q_capacity];
        q_size = 0;
        q_front = -1;
        q_back = -1;
    }

    // Destructor to free dynamically allocated memory
    ~CircularQueue() {
        delete[] q_array;
    }

    // Check if the queue is empty
    bool isEmpty() const {
        return q_size == 0;
    }

    // Check if the queue is full
    bool isFull() const {
        return q_size == q_capacity;
    }

    // Get the size of the queue
    int size() const {
        return q_size;
    }

    // Enqueue operation: Insert an element into the queue
    void enqueue(int value) {
        if (isFull()) {
            cout << "Queue is full. Cannot enqueue " << value << ".\n";
            return;
        }
        if (isEmpty()) {
            q_front = q_back = 0;
        } else {
            q_back = (q_back + 1) % q_capacity;
        }
        q_array[q_back] = value;
        q_size++;
        cout << "Enqueued " << value << " to the queue.\n";
    }

    // Dequeue operation: Remove an element from the queue
    void dequeue() {
        if (isEmpty()) {
            cout << "Queue is empty. Cannot dequeue.\n";
            return;
        }
        cout << "Dequeued " << q_array[q_front] << " from the queue.\n";
        if (q_front == q_back) {  // Queue had one element
            q_front = q_back = -1;
        } else {
            q_front = (q_front + 1) % q_capacity;
        }
        q_size--;
    }

    // Get the front element
    int front() const {
        if (isEmpty()) {
            cout << "Queue is empty.\n";
            return -1;  // Return -1 if empty
        }
        return q_array[q_front];
    }

    // Get the back element
    int back() const {
        if (isEmpty()) {
            cout << "Queue is empty.\n";
            return -1;  // Return -1 if empty
        }
        return q_array[q_back];
    }

    // Clear the queue
    void clear() {
        q_size = 0;
        q_front = q_back = -1;
        cout << "Queue cleared.\n";
    }
```

STDIN

Input for the program (Optional)

Output:

```
Enqueued 10 to the queue.
Enqueued 20 to the queue.
Enqueued 30 to the queue.
Enqueued 40 to the queue.
Enqueued 50 to the queue.
Front: 10
Back: 50
Size: 5
Dequeued 10 from the queue.
Dequeued 20 from the queue.
Enqueued 60 to the queue.
Enqueued 70 to the queue.
Front: 30
Back: 70
Size: 5
Queue cleared.
Size after clearing: 0
```

```
    }
};

int main() {
    CircularQueue q(5);  // Create a queue with capacity 5

    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.enqueue(40);
    q.enqueue(50);  // Queue becomes full

    cout << "Front: " << q.front() << "\n";
    cout << "Back: " << q.back() << "\n";
    cout << "Size: " << q.size() << "\n";

    q.dequeue();
    q.dequeue();
    q.enqueue(60);  // Add new element after dequeue
    q.enqueue(70);  // Queue is full again

    cout << "Front: " << q.front() << "\n";
    cout << "Back: " << q.back() << "\n";
    cout << "Size: " << q.size() << "\n";

    q.clear();  // Clear the queue
    cout << "Size after clearing: " << q.size() << "\n";

    return 0;
}
```

7. Supplementary Activity

Output:

```
Job ID 1 added by Alice for 5 pages.
Job ID 2 added by Bob for 3 pages.
Job ID 3 added by Charlie for 10 pages.
Job ID 4 added by David for 2 pages.
Processing jobs...
Processing Job ID 1 for user Alice with 5 pages.
Processing Job ID 2 for user Bob with 3 pages.
Processing Job ID 3 for user Charlie with 10 pages.
Processing Job ID 4 for user David with 2 pages.
```

The program outputs confirmation messages for each job added, showing the job ID, the user's name, and the number of pages, which reassures users that their submissions have been successfully recorded. When processing the jobs, the output reveals that they are handled in the exact order they were submitted, adhering to the First-Come, First-Served (FCFS) principle. This sequential processing ensures fairness and prevents confusion, as each job is completed before moving on to the next. Overall, the implementation demonstrates effective job management by clearly tracking submissions and maintaining order, which is crucial for user satisfaction in a printing system.

8. Conclusion

In this activity, I learned the importance of using data structures like queues to manage tasks efficiently, particularly in applications like printing systems where order matters. The procedure demonstrated how to create and manage classes effectively in C++, implementing the First-Come, First-Served principle to ensure fairness in job processing. The supplementary activity of simulating multiple job submissions reinforced the understanding of real-world applications of queues. Overall, I feel I performed well in this activity, successfully implementing the requirements and generating the expected output. However, I could improve by exploring error handling for job submissions and expanding functionality to manage job priorities or cancellations.

9. Assessment Rubric