Activity No. 5	
QUEUES	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 10/07/24**
Section:CPE 21S4	**Date Submitted: 10/07/24**
Name(s): Dominic Joseph P. Virtucio	**Instructor: Ma'am Sayo**

6. Output

Using new code

C/C++

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

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

    std::string a = "Dominic";
    std::string b = "Joseph";
    std::string c = "Yzabelle";
    std::string d = "Loreign";
    std::string e = "Bacon";

    q.push(a);
    q.push(b);
    q.push(c);
    q.push(d);
    q.push(e);

    std::cout << "The students
in the queue are:\n";
    while (!q.empty()) {
        std::cout << q.front()
<< "\n";
        q.pop();
    }

    return 0;
}
```

Output

```
/tmp/7nd1OTZLNY.o
The students in the queue are:
Dominic
Joseph
Yzabelle
Loreign
Bacon
```

Modified code provided in the activity

C/C++

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

void displayQueue(std::queue<std::string> q) {
    std::cout << "Current Queue: ";
    while (!q.empty()) {
        std::cout << q.front() << " ";
        q.pop();
    }
    std::cout << "\n";
}

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

    nameQueue.push("Dominic");
    nameQueue.push("Yzabelle");
    nameQueue.push("Joseph");

    std::cout << "The queue of names is: ";
    displayQueue(nameQueue);

    std::cout << "Is the queue empty? : " <<
(nameQueue.empty() ? "Yes" : "No") << "\n";
    std::cout << "Queue size: " <<
nameQueue.size() << "\n";

    if (!nameQueue.empty()) {
        std::cout << "Front: " <<
nameQueue.front() << "\n";
        std::cout << "Back: " <<
nameQueue.back() << "\n";
    }

    nameQueue.pop();
    std::cout << "After popping the first name:
";
    displayQueue(nameQueue);

    nameQueue.push("Loreign");
```

```
    std::cout << "After adding 'Loreign': ";
    displayQueue(nameQueue);

    return 0;
}
```

Output

```
/tmp/RuN3zJsF5e.o
The queue of names is: Current Queue: Dominic Yzabelle Joseph
Is the queue empty? : No
Queue size: 3
Front: Dominic
Back: Joseph
After popping the first name: Current Queue: Yzabelle Joseph
After adding 'Danica': Current Queue: Yzabelle Joseph Loreign
```

The code creates a queue called 'q', inserts strings into it, and outputs each element in the order they were added, exhibiting the First-In, First-Out (FIFO) concept. It has a 'displayQueue' function that produces formatted output and demonstrates different queue operations such as checking for empty, determining size, accessing entries, and changing the queue. Overall, this example efficiently demonstrates queues' essential features in C++, as well as their adaptability in data flow management.

Table 5-1. Queues using C++ STL

C/C++

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

class Node {
public:
    std::string data;
    Node* next;

    Node(const std::string& data) : data(data),
next(nullptr) {}
};

class LinkedListQueue {
public:
    LinkedListQueue() : front(nullptr), rear(nullptr) {}

    void enqueue(const std::string& value) {
        Node* newNode = new Node(value);
        if (isEmpty()) {
            front = rear = newNode;
        } else {
            rear->next = newNode;
            rear = newNode;
        }
        std::cout << value << " added to the queue.\n";
    }

    void dequeue() {
        if (isEmpty()) {
            std::cout << "Queue is empty. Cannot
dequeue.\n";
            return;
        }
        Node* temp = front;
        std::cout << front->data << " removed from the
queue.\n";
        front = front->next;
        delete temp;

        if (front == nullptr) {
            rear = nullptr;
        }
    }

    void display() const {
        if (isEmpty()) {
            std::cout << "The queue is empty.\n";
            return;
        }
        Node* current = front;
        std::cout << "The students in the queue are:\n";
        while (current) {
            std::cout << current->data << "\n";
            current = current->next;
        }
    }

    bool isEmpty() const {
        return front == nullptr;
    }
```

Output

```
/tmp/aT4ajoH6aI.o
Dominic added to the queue.
Joseph added to the queue.
Yzabelle added to the queue.
Loreign added to the queue.
Bacon added to the queue.
The students in the queue are:
Dominic
Joseph
Yzabelle
Loreign
Bacon
Dominic removed from the queue.
The students in the queue are:
Joseph
Yzabelle
Loreign
Bacon
Joseph removed from the queue.
The students in the queue are:
Yzabelle
Loreign
Bacon
Yzabelle removed from the queue.
The students in the queue are:
Loreign
Bacon
Loreign removed from the queue.
The students in the queue are:
Bacon
```

Observation:

This C++ code builds a queue data structure using a linked list, with a Node class that stores data and a reference to the next node, as well as a LinkedListQueue class that manages enqueueing, dequeuing, and displaying the items while properly managing memory. The main function illustrates the capability by enqueuing five strings, showing the queue, and then dequeuing components until the queue is empty, guaranteeing that the FIFO principle is followed. Overall, the implementation demonstrates dynamic scaling, rudimentary error handling for empty queues, and an accurate output representation of the

`` private: Node* front; Node* rear; }; int main() { LinkedListQueue queue; queue.enqueue("Dominic"); queue.enqueue("Joseph"); queue.enqueue("Yzabelle"); queue.enqueue("Loreign"); queue.enqueue("Bacon"); queue.display(); queue.dequeue(); queue.display(); queue.dequeue(); queue.display(); queue.dequeue(); queue.display(); queue.dequeue(); queue.display(); return 0; } ```	**queue's contents.**

Table 5-2. Queues using Linked List Implementation

C/C++

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

template <typename T>
class Queue {
private:
    T* q_array;
    int q_front, q_back, q_size, q_capacity;

public:
    Queue(int capacity = 10) : q_capacity(capacity), q_size(0), q_front(0), q_back(0) {
        q_array = new T[q_capacity];
    }

    ~Queue() {
        delete[] q_array;
```

```
    }

    bool isEmpty() const { return q_size == 0; }
    int size() const { return q_size; }

    void enqueue(const T& item) {
        if (q_size == q_capacity) {
            std::cerr << "Queue Overflow!" << std::endl;
            return;
        }
        q_array[q_back] = item;
        q_back = (q_back + 1) % q_capacity;
        q_size++;
    }

    T dequeue() {
        if (isEmpty()) {
            std::cerr << "Queue Underflow!" << std::endl;
            return T();
        }
        T frontItem = q_array[q_front];
        q_front = (q_front + 1) % q_capacity;
        q_size--;
        return frontItem;
    }

    T front() const {
        if (isEmpty()) {
            std::cerr << "Queue is empty!" << std::endl;
            return T();
        }
        return q_array[q_front];
    }
};

int main() {
    Queue<int> myQueue(5);
    myQueue.enqueue(10);
    myQueue.enqueue(20);
    myQueue.enqueue(30);
    std::cout << "Queue size: " << myQueue.size() << "\nFront element: " <<
myQueue.front() << "\nDequeued element: " << myQueue.dequeue() << "\nQueue size: " <<
myQueue.size() << std::endl;
    return 0;
}
```

Output

```
/tmp/oQFlNhIqDR.o
Queue size: 3
Front element: 10
Dequeued element: 10
Queue size: 2
```

observation:
This C++ code builds a generic queue using an array, following the First-In, First-Out (FIFO) principle and including important operations like as 'enqueue', 'dequeue', and front retrieval. It contains dynamic memory management and simple error handling for overflow and underflow circumstances, assuring resource efficiency via a circular queue system. Overall, it is a useful example of a queue solution, displaying both template flexibility and efficient memory management.

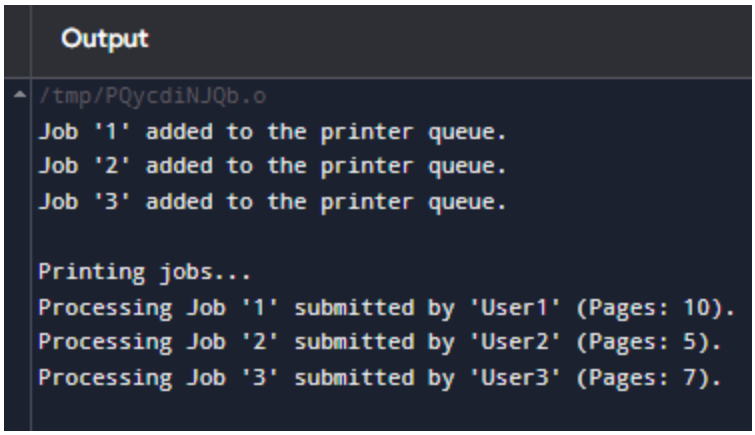
Table 5-3. Queues using Array Implementation

7. Supplementary Activity

C/C++

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

// Step 1: Create a class called Job
class Job {
public:
    Job(int id, const std::string& userName, int pages)
        : id(id), userName(userName), pages(pages) {}

    int getId() const { return id; }
    const std::string& getUserName() const { return userName; }
    int getPages() const { return pages; }

private:
    int id;
    std::string userName;
    int pages;
};

// Step 2: Create a class called Printer
class Printer {
public:
    Printer(int capacity) : capacity(capacity), front(0), rear(-1), size(0) {
        jobsQueue = new Job*[capacity];
    }

    ~Printer() {
        delete[] jobsQueue;
    }

    void addJob(const Job& job) {
        if (isFull()) {
            std::cout << "Printer queue is full. Job '" << job.getId() << "' cannot be
added.\n";
        } else {
            rear = (rear + 1) % capacity;
            jobsQueue[rear] = new Job(job);
            size++;
            std::cout << "Job '" << job.getId() << "' added to the printer queue.\n";
        }
    }
```

```
    void processJobs() {
        while (!isEmpty()) {
            Job* currentJob = jobsQueue[front];
            front = (front + 1) % capacity;
            size--;

            std::cout << "Processing Job '" << currentJob->getId() << "' submitted by '"
                      << currentJob->getUserName() << "' (Pages: " <<
currentJob->getPages() << ").\n";

            delete currentJob;
        }
    }

    bool isEmpty() const {
        return size == 0;
    }

    bool isFull() const {
        return size == capacity;
    }

private:
    int capacity;
    int front;
    int rear;
    int size;
    Job** jobsQueue;
};

int main() {
    // Step 4: Simulate a scenario with multiple job submissions
    Printer printer(5);

    printer.addJob(Job(1, "User1", 10));
    printer.addJob(Job(2, "User2", 5));
    printer.addJob(Job(3, "User3", 7));

    // Step 3: Process all the pending jobs
    std::cout << "\nPrinting jobs...\n";
    printer.processJobs();

    return 0;
}
```



8. Conclusion

This project provided me with a better knowledge of queues in C++, since I looked at how they were implemented using the C++ STL as well as custom linked lists and arrays. I studied essential queue functions including 'enqueue', 'dequeue', 'front', 'isEmpty', and'size' and weighed the benefits and drawbacks of each implementation type. The printer simulation demonstrated the practical uses of queues in handling jobs in a first-in, first-out manner, emphasizing their importance in resource management and order processing in software development.

9. Assessment Rubric