Activity No. <5.1>	
<Queue – Linked List Application>	
Course Code: CPE010	**Program: BSCPE**
Course Title: Data Structures and Algorithms	**Date Performed: 9/9/2025**
Section: CPE21S4	**Date Submitted: 9/9/2025**
Name(s): Anastacio, Lester Arvid P.	**Instructor: Engr. Jimlord Quejado**
6. Output	

CODE SCREENSHOTS

MAIN FILE:

Main.cpp Qheader.h

```cpp
#include <iostream>
#include "Qheader.h"

int main(){
    Queue<std::string> CPE21S4;

    CPE21S4.enqueue("Francis");
    CPE21S4.enqueue("JASON!!!");
    CPE21S4.enqueue("Curwin");
    CPE21S4.enqueue("ABILA");
    CPE21S4.enqueue("Dano");
    CPE21S4.getFront();

    CPE21S4.dequeue();
    CPE21S4.getFront();
    CPE21S4.getRear();

    CPE21S4.Display();


    return 0;
}
```

ANALYSIS:

In this main file:

Queue<std::string> CPE21S4; = it is the one that creates the queue object in which stores strings.
CPE21S4.enqueue("temp"); = this function adds in the objects that will be added to the queue.
CPE21S4.dequeue(); = this function removes the first element within the queue.
CPE21S4.getFront(); = this one gets the element on the front of the queue.
CPE21S4.getRear(); = this one gets the last element which is in the rear of the queue
CPE21S4.Display(); = this one display all the current elements within the queue.

HEADER FILE:

Main.cpp Qheader.h

```cpp
#ifndef QHEADER_H
#define QHEADER_H
#include <iostream>
template<typename T>

class Node{
    public:
        T data;
        Node* next;

        Node(T new_data){
            data = new_data;
            next = nullptr;

        }

};

template<typename T>
class Queue{
    private:
        Node<T> *front;
        Node<T> *rear;

    public:
        // Create an empty queue
        Queue(){
            front = rear = nullptr;
            std::cout << "A Queue has been created.."<<std::endl;
        }


        //isEmpty
        bool isEmpty(){
            return front == nullptr;
        }

        //enqueue
        void enqueue(T new_data){
            Node<T> *new_node  = new Node<T> (new_data);

            if (isEmpty()){
                front = rear = new_node;
                std::cout<< "Enqueue to an empty queue"<<std::endl;

                return;
            }
            rear->next = new_node;
            rear = new_node;
            std::cout<< "Successfully Enqueued. "<<std::endl;

        }
```

```cpp

        //dequeue
        void dequeue(){
            if (isEmpty()){
                std::cout<<"The Queue is Empty"<<std::endl;
                return;
            }

            //storing the front to a temporary pointer
            Node<T>* temp = front;

            //check if after the dequeue, the queue is empty
            if (front == nullptr){
                rear == nullptr;
            }
            else{
                //reassign the front to the next node
                front = front->next;
            }

            delete temp;

        }

        //getfront
        void getFront(){
            if (isEmpty()){
                std::cout<<"The Queue is Empty"<<std::endl;
                return;
            }

            std::cout<<"Current Front: "<< front -> data << std::endl;

        }

        //getrear
        void getRear(){
            if (isEmpty()){
                std::cout<<"The Queue is Empty"<<std::endl;
                return;
            }

            std::cout<<"Current Rear: "<< rear -> data << std::endl;
        }

        //display
        void Display(){
            if (isEmpty()){
                std::cout<<"The Queue is Empty"<<std::endl;
                return;
            }

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




#endif
```

```
111         }
112
113         //to deallocate memory
114         ~Queue(){
115             while(!isEmpty()){
116                 dequeue();
117             }
118         }
119 };
120
121
```

ANALYSIS:
In this header file:

Code:

```
template<typename T>
class Node {
public:
    T data;
    Node* next;

    Node(T new_data) {
        data = new_data;
        next = nullptr;
    }
};
```

Analysis: this function is the building block of the queue, it stores data types of T and a pointer for the next node

Code:

```
template<typename T>
class Queue {
private:
    Node<T> *front;
    Node<T> *rear;
```

Analysis: **this function stores a class which is names Queue and within it is the Nodes for the front and rear, the pointers here manage the queue's boundaries.**

Code:

```
Queue() {
    front = rear = nullptr;
    std::cout << "A Queue has been created.." << std::endl;
}
```

Analysis: **This is the constructor of the code, this function initializes an empty queue within this also confirms whether the a queue is actually created.**

Code:

```
bool isEmpty() {
```

```
    return front == nullptr;
}
```

Analysis: This functions checks whether the queue is empty by verifying whether the front is equals to nullptr.

Code: ENQUEUE

```
void enqueue(T new_data) {
    Node<T> *new_node = new Node<T>(new_data);

    if (isEmpty()) {
        front = rear = new_node;
        std::cout << "Enqueue to an empty queue" << std::endl;
        return;
    }

    rear->next = new_node;
    rear = new_node;
    std::cout << "Successfully Enqueued." << std::endl;
}
```

Analysis: In this function, this adds a new node to the rear of the queue as stated by the code rear -> = new_node, within this function also checks whether the queue is empty and will print out an message corresponds whether a new node is added or empty.

Code: DEQUEUE

```
void dequeue() {
    if (isEmpty()) {
        std::cout << "The Queue is Empty" << std::endl;
        return;
    }

    Node<T>* temp = front;

    if (front == nullptr) {
        rear == nullptr;
    } else {
        front = front->next;
    }

    delete temp;
}
```

Analysis: This function removes the front node, in this function also exists the is the checking whether the queue is empty or not, and Node<T>* temp = front; here acts as a temporary post so that the perimeter of the front wont be deleted alongside the first element.

Code: getFront

```
void getFront() {
    if (isEmpty()) {
```

```
        std::cout << "The Queue is Empty" << std::endl;
        return;
    }

    std::cout << "Current Front: " << front->data << std::endl;
}
```

Analysis: This function displays the data at the front of the queue.

Code: getRear

```
void getRear() {
    if (isEmpty()) {
        std::cout << "The Queue is Empty" << std::endl;
        return;
    }

    std::cout << "Current Rear: " << rear->data << std::endl;
}
```

Analysis: this function displays the data at the rear of the queue.

Code: Display

```
void Display() {
    if (isEmpty()) {
        std::cout << "The Queue is Empty" << std::endl;
        return;
    }

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

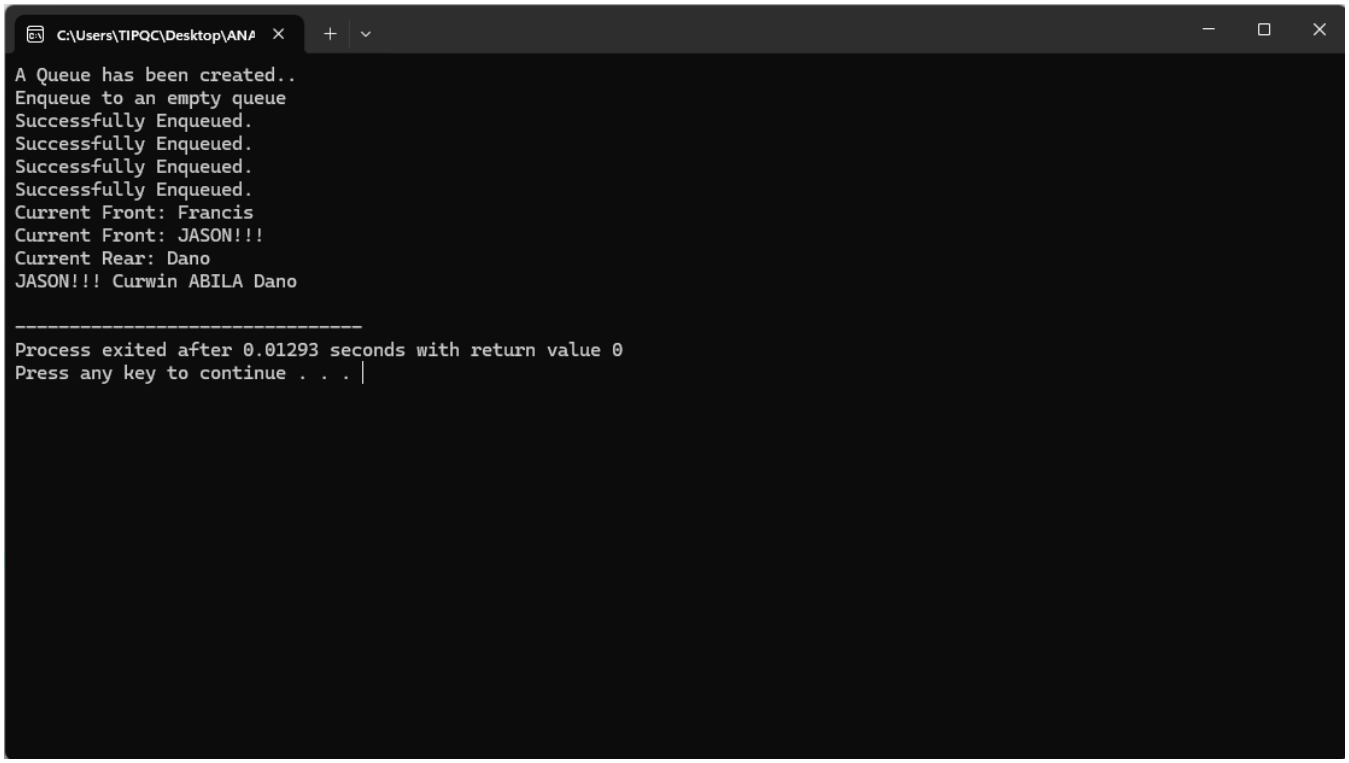
Analysis: This function displays all the elements within the queue by traversing the queue from front to the rear and prints out each element.

Code:

```
~Queue() {
    while (!isEmpty()) {
        dequeue();
    }
}
```

Analysis: This is the deconstructor of the code, this function ensures all dynamically allocated memory is freed whenever the queue object is destroyed.

OUTPUT:



7. Supplementary Activity

8. Conclusion

During this practice activity, I've learned the step by step process of creating and knowing how the Queue using link list works, even though it is a practice activity it still prove a bit challenging and confusing at first, but after some bit I finally understood how the enqueue and dequeue works and the other functions needed for the whole code to work.

9. Assessment Rubric