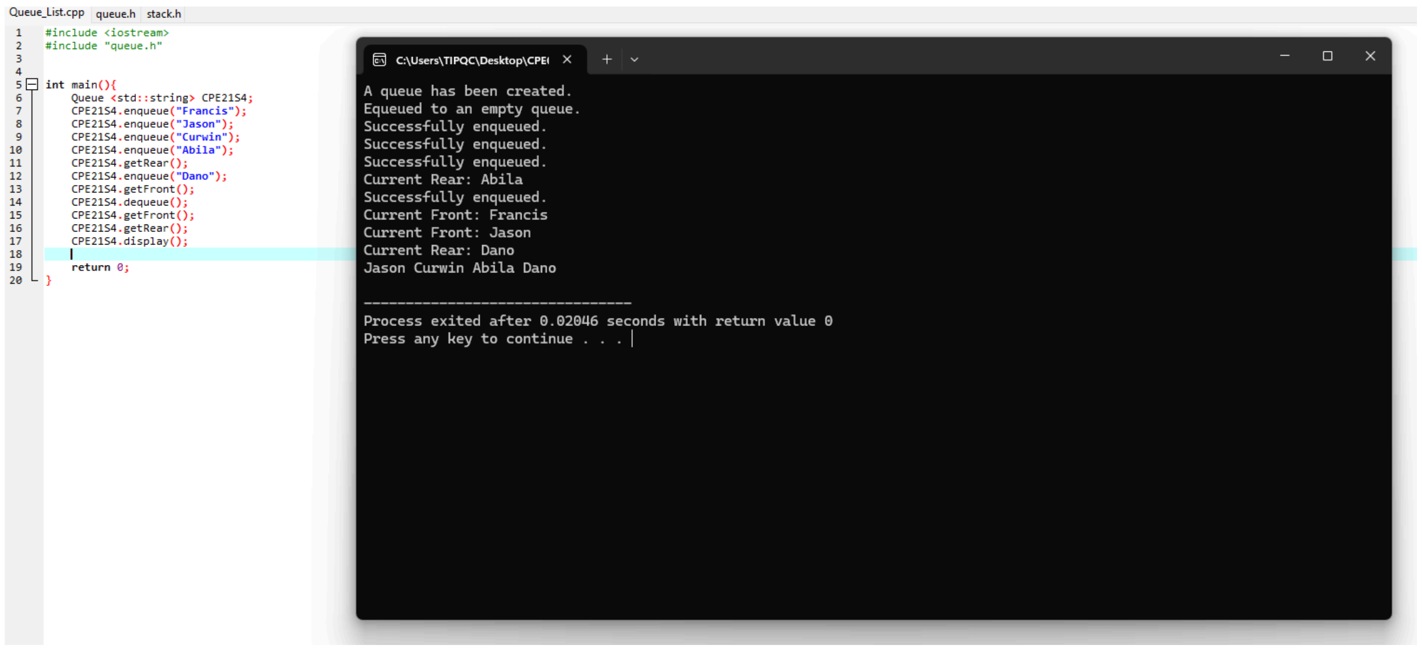
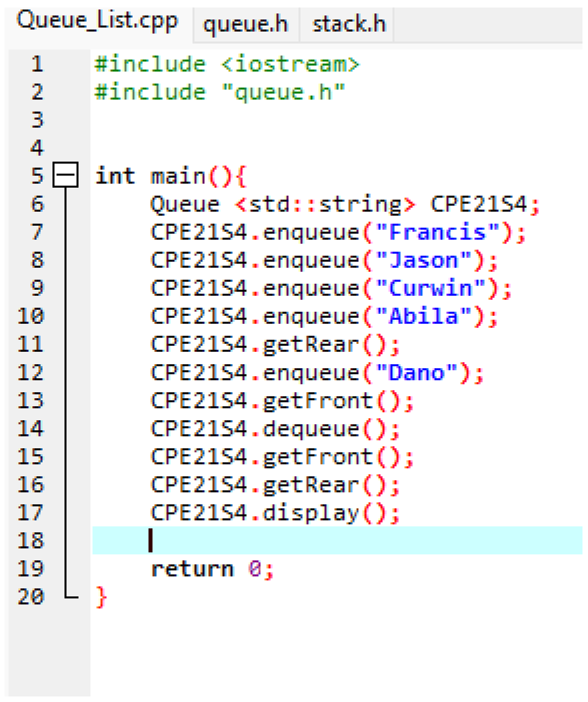
Seatwork 5.1	
Queue - Linked List Application	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 09/09/2025
Section: CPE21S4	**Date Submitted:** 09/09/2025
Name(s): Cabrera, Gabriel A.	**Instructor:** Engr. Jimlord Quejado

6. Output

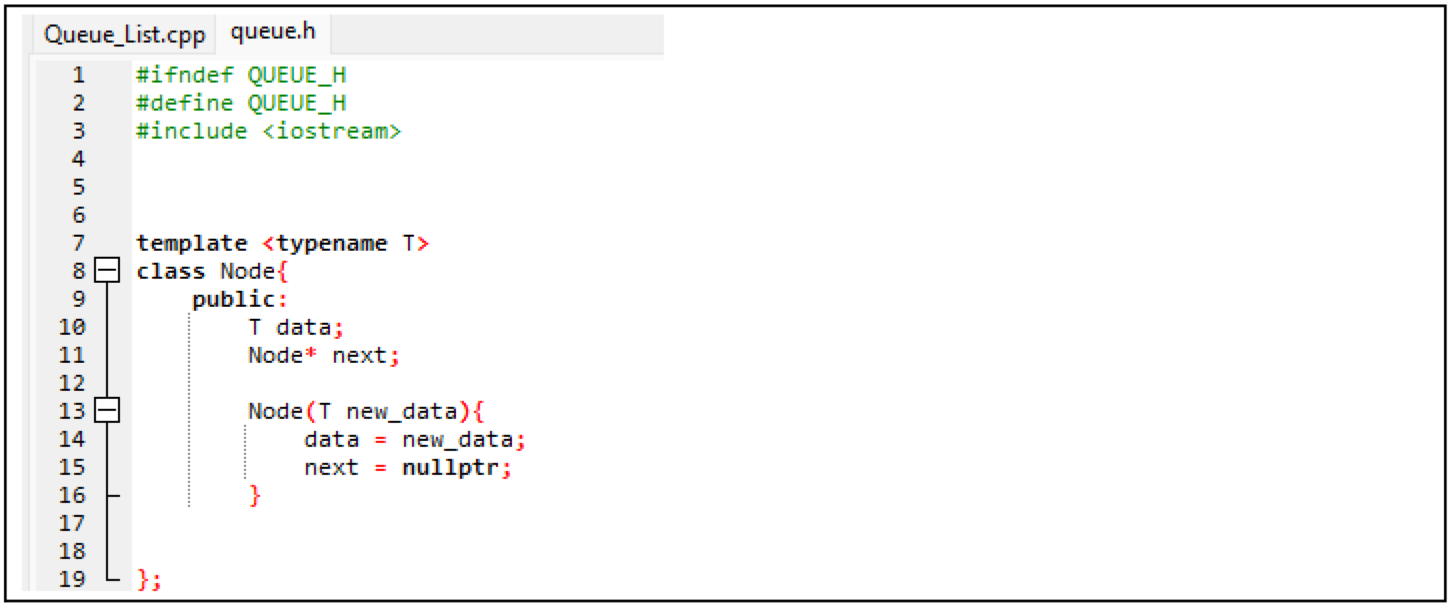


```cpp
#include <iostream>
#include "queue.h"


int main(){
	Queue <std::string> CPE21S4;
	CPE21S4.enqueue("Francis");
	CPE21S4.enqueue("Jason");
	CPE21S4.enqueue("Curwin");
	CPE21S4.enqueue("Abila");
	CPE21S4.getRear();
	CPE21S4.enqueue("Dano");
	CPE21S4.getFront();
	CPE21S4.dequeue();
	CPE21S4.getFront();
	CPE21S4.getRear();
	CPE21S4.display();
	
	return 0;
}
```

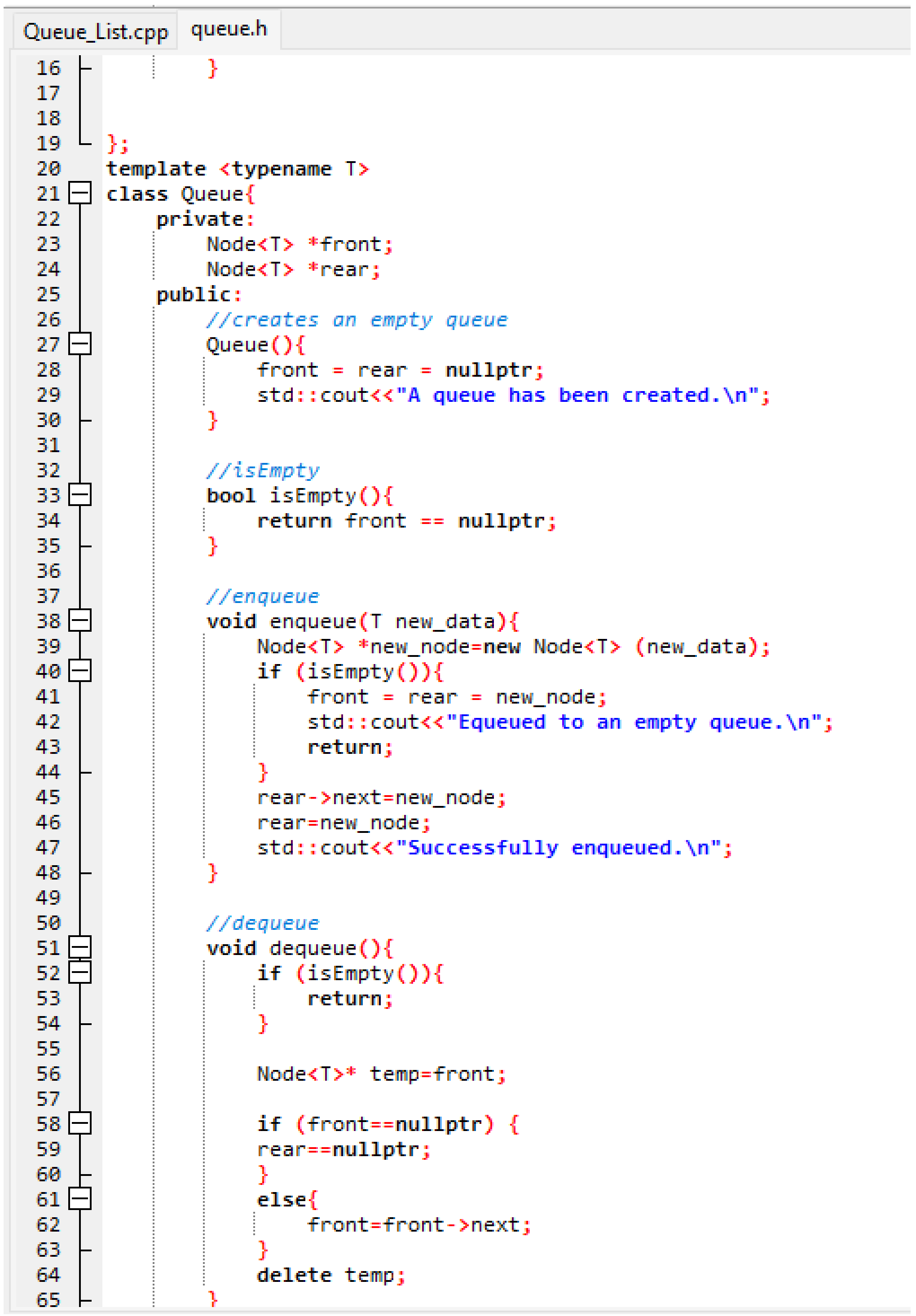
```
A queue has been created.
Equeued to an empty queue.
Successfully enqueued.
Successfully enqueued.
Successfully enqueued.
Current Rear: Abila
Successfully enqueued.
Current Front: Francis
Current Front: Jason
Current Rear: Dano
Jason Curwin Abila Dano

--------------------------------
Process exited after 0.02046 seconds with return value 0
Press any key to continue . . .
```



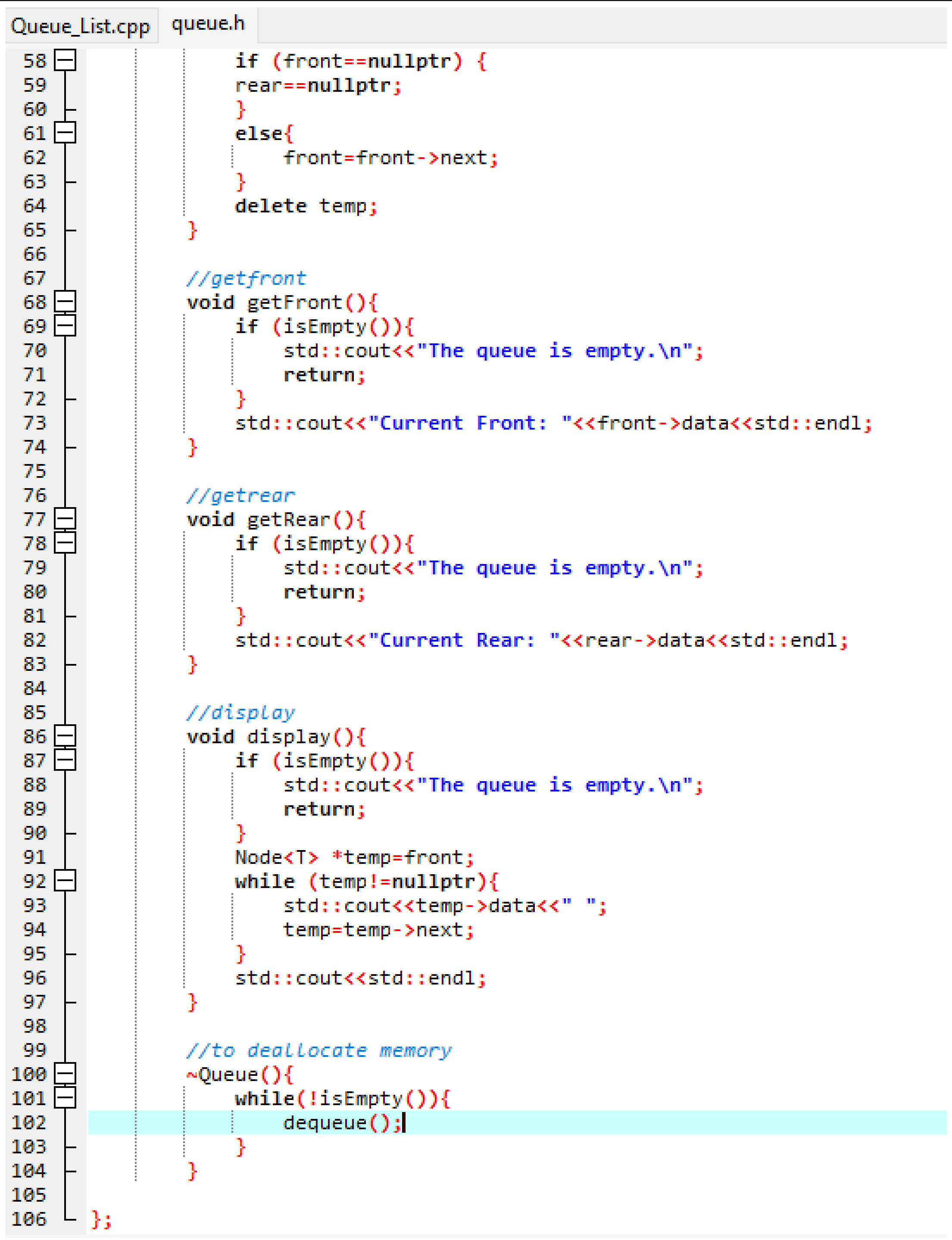
```cpp
#include <iostream>
#include "queue.h"


int main(){
	Queue <std::string> CPE21S4;
	CPE21S4.enqueue("Francis");
	CPE21S4.enqueue("Jason");
	CPE21S4.enqueue("Curwin");
	CPE21S4.enqueue("Abila");
	CPE21S4.getRear();
	CPE21S4.enqueue("Dano");
	CPE21S4.getFront();
	CPE21S4.dequeue();
	CPE21S4.getFront();
	CPE21S4.getRear();
	CPE21S4.display();
	
	return 0;
}
```



```cpp
#ifndef QUEUE_H
#define QUEUE_H
#include <iostream>



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

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


};
```


Queue_List.cpp | queue.h

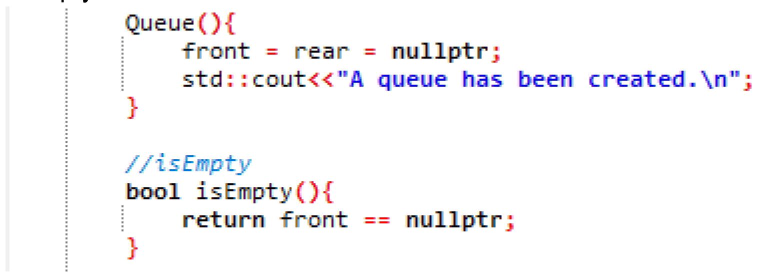
```cpp
		}


};
template <typename T>
class Queue{
	private:
		Node<T> *front;
		Node<T> *rear;
	public:
		//creates an empty queue
		Queue(){
			front = rear = nullptr;
			std::cout<<"A queue has been created.\n";
		}

		//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<<"Equeued to an empty queue.\n";
				return;
			}
			rear->next=new_node;
			rear=new_node;
			std::cout<<"Successfully enqueued.\n";
		}

		//dequeue
		void dequeue(){
			if (isEmpty()){
				return;
			}

			Node<T>* temp=front;

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



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

        //getfront
        void getFront(){
            if (isEmpty()){
                std::cout<<"The queue is empty.\n";
                return;
            }
            std::cout<<"Current Front: "<<front->data<<std::endl;
        }

        //getrear
        void getRear(){
            if (isEmpty()){
                std::cout<<"The queue is empty.\n";
                return;
            }
            std::cout<<"Current Rear: "<<rear->data<<std::endl;
        }

        //display
        void display(){
            if (isEmpty()){
                std::cout<<"The queue is empty.\n";
                return;
            }
            Node<T> *temp=front;
            while (temp!=nullptr){
                std::cout<<temp->data<<" ";
                temp=temp->next;
            }
            std::cout<<std::endl;
        }

        //to deallocate memory
        ~Queue(){
            while(!isEmpty()){
                dequeue();
            }
        }

};
```

7. Supplementary Activity

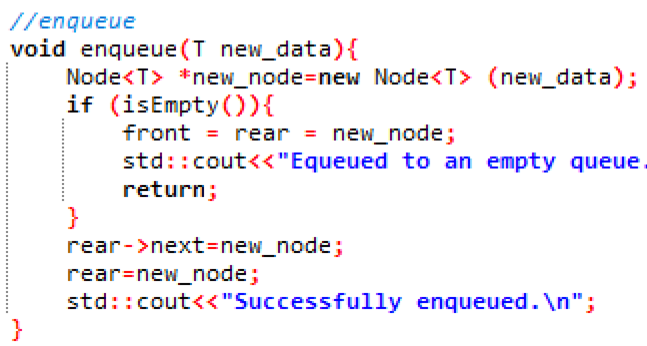
a. isEmpty



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

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

The Queue class first started to make an empty list so that there’s a placeholder for the empty data and pointers, declared as front and rear, for the queue object to be made or filled. The isEmpty function just checks if the front data is still assigned to a null pointer, determining that the queue is empty. The function itself just returns a 1 or 0 depending whether the queue is empty or not and we can use this on the future functions to output an error or a validation statement.

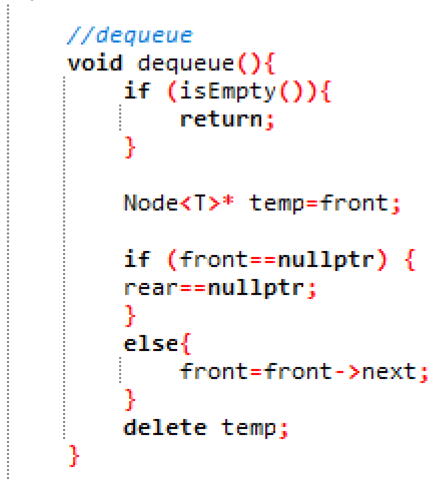
b. Enqueue



```
//enqueue
void enqueue(T new_data){
    Node<T> *new_node=new Node<T> (new_data);
    if (isEmpty()){
        front = rear = new_node;
        std::cout<<"Equeued to an empty queue.\n";
        return;
    }
    rear->next=new_node;
    rear=new_node;
    std::cout<<"Successfully enqueued.\n";
}
```

The function first uses the previously made Node class to make an object reference named new_node, this new node will refer to the parameter entered in the function which is the new_data, using the type T so that the function can work on any type of variable entered. After making the new node, the function uses the isEmpty statement to check if the queue is previously empty. If the isEmpty function returns true, this means that the enqueued data is the first element in the queue, the program will then make the new data entered as its front and rear. The program will then output a validation statement and end the function. If the isEmpty function returns a false, it will move on to the next part of the function, which assigns the rear variable and the rear’s next, which is its pointer, to the new node. The function ends by printing a validation statement that says the data is successfully enqueued.

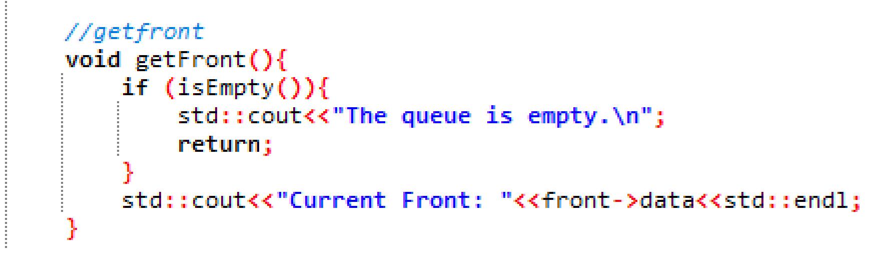
c. Dequeue



```
//dequeue
void dequeue(){
    if (isEmpty()){
        return;
    }

    Node<T>* temp=front;

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

Like the previous function, the function will first check if the queue is already empty, if it is, then it will simply stop the function. Otherwise the function will continue and start by creating a new Node object pointer named temp and assign it to the front's address, this is so it would not lose the front's address when the program reassigns it. The function would recheck if the queue will be emptied by doing so, if it is, then it would assign the rear too to the null pointer, essentially emptying the queue completely. If the queue would not be emptied, the front data will be assigned to the next node's data. The function would finish by deleting the temp variable.

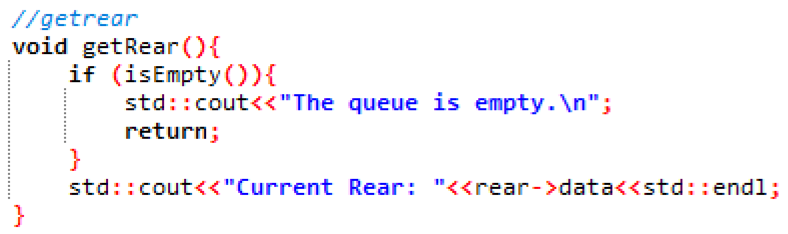
d. getFront



```
//getfront
void getFront(){
    if (isEmpty()){
        std::cout<<"The queue is empty.\n";
        return;
    }
    std::cout<<"Current Front: "<<front->data<<std::endl;
}
```

The function first checks if the queue is empty first by using the isEmpty function, if it is, then it would display a validation sentence ("The queue is empty."). If the queue is not empty, the program will display the current front of the queue by getting the front node's data.

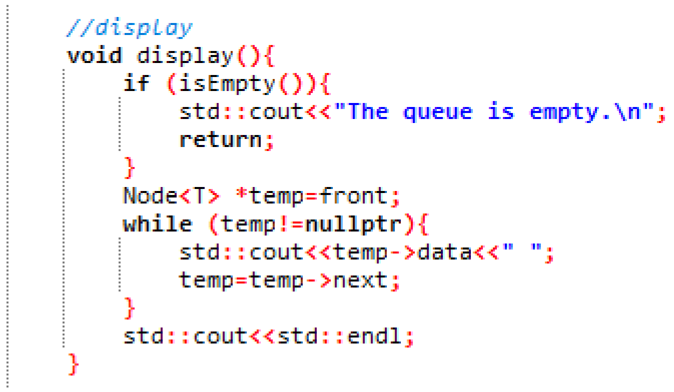
e. getRear



```
//getrear
void getRear(){
    if (isEmpty()){
        std::cout<<"The queue is empty.\n";
        return;
    }
    std::cout<<"Current Rear: "<<rear->data<<std::endl;
}
```

Very similar to the previous function, this one also checks if the queue is empty and would display a validation statement if it is. Otherwise it would display the current rear node's data by using the rear-> data function.

f. Display



```
//display
void display(){
    if (isEmpty()){
        std::cout<<"The queue is empty.\n";
        return;
    }
    Node<T> *temp=front;
    while (temp!=nullptr){
        std::cout<<temp->data<<" ";
        temp=temp->next;
    }
    std::cout<<std::endl;
}
```

Like all the other functions, it checks if the queue is empty first and displays a statement if it is. If the program has determined that the queue is not empty, it would assign a temp data pointer to the front of the queue, it would then start looping on printing the temp data and going to the next node until the temp pointer points to nullptr, which means that the pointer already reached the end of the queue and is done displaying all the element in the queue.

8. Conclusion

The activity has successfully shown how to implement the queue function using a linked list. This also broadened my knowledge and understanding about linked list too, I would surely use this activity to learn more and implement in more future programs.

9. Assessment Rubric