

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
1  #include <iostream>
2  #include "Qheader.h"
3
4  int main(){
5      Queue<std::string> CPE21S4;
6
7      CPE21S4.enqueue("Francis");
8      CPE21S4.enqueue("JASON!!!");
9      CPE21S4.enqueue("Curwin");
10     CPE21S4.enqueue("ABILA");
11     CPE21S4.enqueue("Dano");
12     CPE21S4.getFront();
13
14     CPE21S4.dequeue();
15     CPE21S4.getFront();
16     CPE21S4.getRear();
17
18     CPE21S4.Display();
19
20
21     return 0;
22 }

```

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:

```
1  #ifndef QHEADER_H
2  #define QHEADER_H
3  #include <iostream>
4  template<typename T>
5
6  class Node{
7  public:
8      T data;
9      Node* next;
10
11      Node(T new_data){
12          data = new_data;
13          next = nullptr;
14      }
15  };
16
17
18
19 template<typename T>
20 class Queue{
21 private:
22     Node<T> *front;
23     Node<T> *rear;
24
25 public:
26     // Create an empty queue
27     Queue(){
28         front = rear = nullptr;
29         std::cout << "A Queue has been created.."<<std::endl;
30     }
31
32
33     //isEmpty
34     bool isEmpty(){
35         return front == nullptr;
36     }
37
38     //enqueue
39     void enqueue(T new_data){
40         Node<T> *new_node = new Node<T> (new_data);
41
42         if (isEmpty()){
43             front = rear = new_node;
44             std::cout<< "Enqueue to an empty queue"<<std::endl;
45
46             return;
47         }
48         rear->next = new_node;
49         rear = new_node;
50         std::cout<< "Successfully Enqueued. " <<std::endl;
51     }
52 }
```

```

53
54 //dequeue
55 void dequeue(){
56     if (isEmpty()){
57         std::cout<<"The Queue is Empty"<<std::endl;
58         return;
59     }
60
61     //storing the front to a temporary pointer
62     Node<T>* temp = front;
63
64     //check if after the dequeue, the queue is empty
65     if (front == nullptr){
66         rear = nullptr;
67     }
68     else{
69         //reassign the front to the next node
70         front = front->next;
71     }
72
73     delete temp;
74
75 }
76
77 //getfront
78 void getFront(){
79     if (isEmpty()){
80         std::cout<<"The Queue is Empty"<<std::endl;
81         return;
82     }
83
84     std::cout<<"Current Front: "<< front -> data << std::endl;
85
86 }
87
88 //getrear
89 void getRear(){
90     if (isEmpty()){
91         std::cout<<"The Queue is Empty"<<std::endl;
92         return;
93     }
94
95     std::cout<<"Current Rear: "<< rear -> data << std::endl;
96
97 }
98
99 //display
100 void Display(){
101     if (isEmpty()){
102         std::cout<<"The Queue is Empty"<<std::endl;
103         return;
104     }
105
106     Node<T> *temp = front;
107     while (temp !=nullptr){
108         std::cout<< temp -> data << " ";
109         temp = temp -> next;
110     }
111     std::cout<<std::endl;
112 };
113
114
115
116
117
118
119 #endif

```

```

111 | }
112 |
113 | //to deallocate memory
114 | ~Queue(){
115 |     while(!isEmpty()){
116 |         dequeue();
117 |     }
118 | }
119 | };
120 |
121 |
122 |

```

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 function checks whether the queue is empty by verifying whether the front is equal 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 a message corresponding 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 checking whether the queue is empty or not, and `Node<T>* temp = front`; here acts as a temporary pointer so that the pointer of the front won't 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 destructor of the code, this function ensures all dynamically allocated memory is freed whenever the queue object is destroyed.

OUTPUT:

```
C:\Users\TIPOC\Desktop\ANA X + v
A Queue has been created..
Enqueue to an empty queue
Successfully Enqueued.
Successfully Enqueued.
Successfully Enqueued.
Successfully Enqueued.
Current Front: Francis
Current Front: JASON!!!
Current Rear: Dano
JASON!!! Curwin ABILA Dano

-----
Process exited after 0.01293 seconds with return value 0
Press any key to continue . . . |
```

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

--