Week4_Queue Implementation

Scenario:
A **Cafeteria line order,** multiple guests need to be in line to order their food & drinks. Seller serves food to our customers by the order ID which makes them feel fairly treated. More customers who come later must be at the back of the line and the front must serve his/her food first. Order ID increases ascendingly.

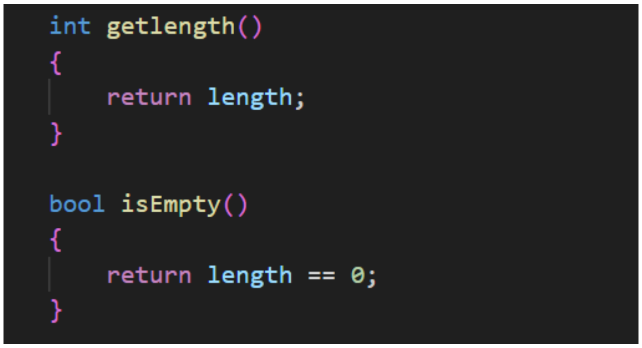
Implementation Type: Linked List Queue
Reason:

- Number of customers each time is not fixed, dynamically changing
- No need compaction
- Most operations are O(1)

Diagram:
front-> [Order01]-> [Order02]-> - - - -> [OrderN] <-rear
            ^ dequeue   ^ enqueue

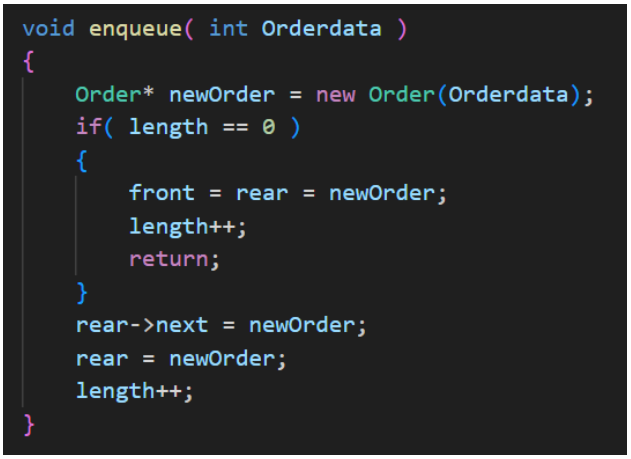
Methods:

- getlength: Return number of order.
- isEmpty: Ensure we have orders when we need to remove any order from queue.



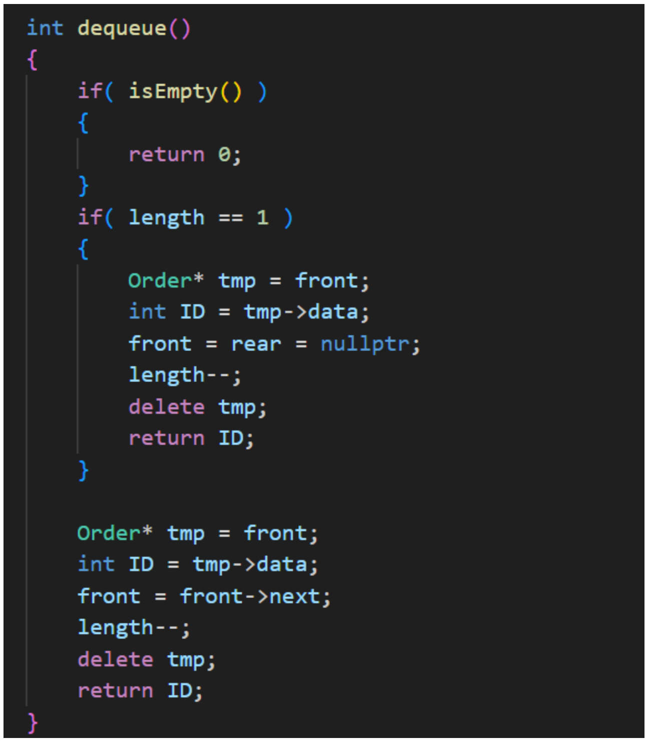
```
int getlength()
{
    return length;
}

bool isEmpty()
{
    return length == 0;
}
```

- print: Show all orders.
- enqueue: add new customer's order to the list at the last to arrange in order



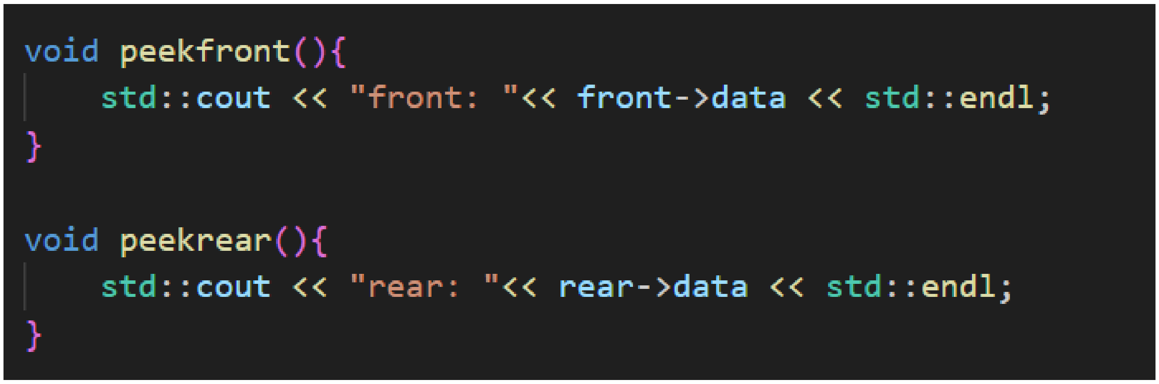
```
void enqueue( int Orderdata )
{
    Order* newOrder = new Order(Orderdata);
    if( length == 0 )
    {
        front = rear = newOrder;
        length++;
        return;
    }
    rear->next = newOrder;
    rear = newOrder;
    length++;
}
```

- dequeue: take out any order to modify and remove it if necessary, but we also check first before the removing avoiding crashing program.



```
int dequeue()
{
    if( isEmpty() )
    {
        return 0;
    }
    if( length == 1 )
    {
        Order* tmp = front;
        int ID = tmp->data;
        front = rear = nullptr;
        length--;
        delete tmp;
        return ID;
    }

    Order* tmp = front;
    int ID = tmp->data;
    front = front->next;
    length--;
    delete tmp;
    return ID;
}
```

- peekfront: Show the first order without taking it out or modifying.
- peekrear: Show the last order without taking it out or modifying.



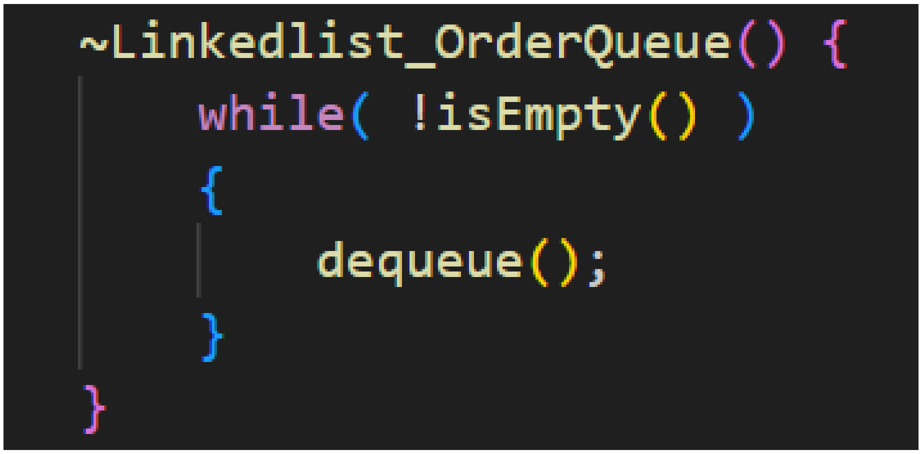
```
void peekfront(){
    std::cout << "front: "<< front->data << std::endl;
}

void peekrear(){
    std::cout << "rear: "<< rear->data << std::endl;
}
```

Special choice:

- Empty Queue: Always detect and alert the user and return nothing.
- First job adds: Front also be a rear.
- Last job removes: Both front and rear are reset to NULL.

Special Method:

- ~Linkedlist_OrderQueue: For handling the memory, this is destructor is for deleting all memory after object is no longer exist. Working with pointers is risky to leak memory.



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

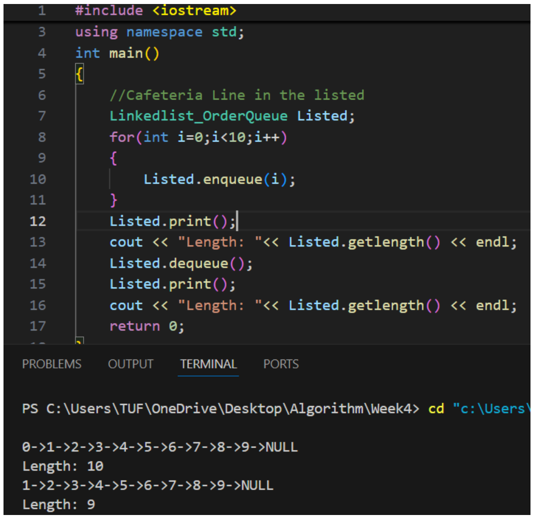
Edge cases handled:

Edge Case	How
Dequeue from empty queue	Return 0 and alert it's empty
Peek from empty queue	Return nothing and alert it's empty
Removing last job	Set front = rear = NULL
Adding first job	Set front = rear = newOrder
Memory leaks	Each removed order is deleted
Dynamic growth	New order created as needed

Time complexity: With Linked list is faster and consistent, almost of the cases are O(1) so the time complexity does not matter, anyway O(n) still occur but it is not necessary use it, just to print the whole queue to see detail.

Evidence of Correctness:

- Each new order enters at the rear (enqueue).
- Each order leaves from the front (dequeue).
  Thus, the first order inserted is always the first to be served.



```cpp
#include <iostream>
using namespace std;
int main()
{
    //Cafeteria Line in the listed
    Linkedlist_OrderQueue Listed;
    for(int i=0;i<10;i++)
    {
        Listed.enqueue(i);
    }
    Listed.print();
    cout << "Length: "<< Listed.getlength() << endl;
    Listed.dequeue();
    Listed.print();
    cout << "Length: "<< Listed.getlength() << endl;
    return 0;
```

```
PS C:\Users\TUF\OneDrive\Desktop\Algorithm\Week4> cd "c:\Users\

0->1->2->3->4->5->6->7->8->9->NULL
Length: 10
1->2->3->4->5->6->7->8->9->NULL
Length: 9
```

Sample Test Run