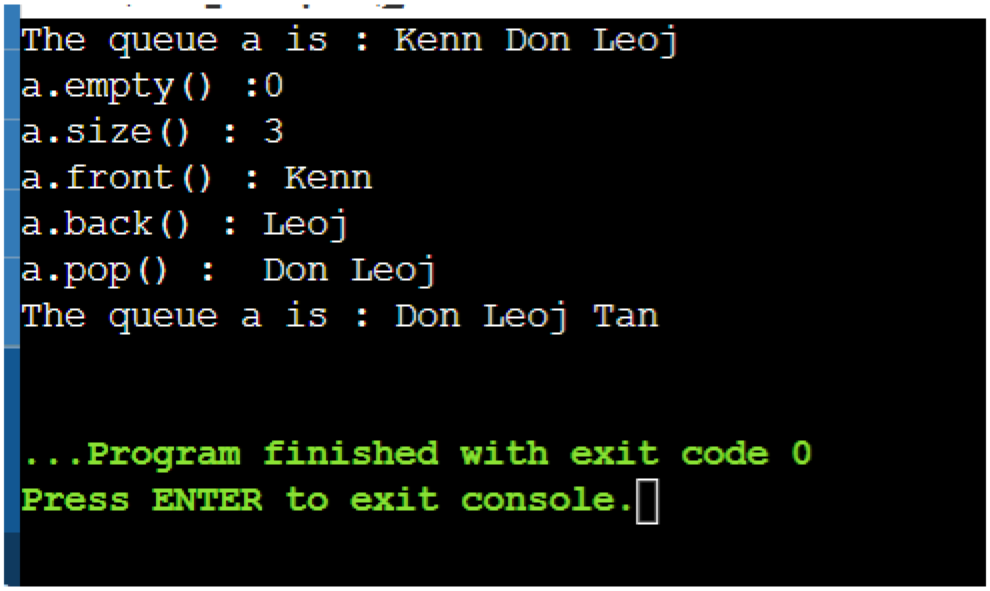
Activity No. 5	
Hands-on Activity 5.1 Queues	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 07/10/2024**
Section: CPE21S4	**Date Submitted: 07/10/2024**
Name(s): Kenn Jie Valleser	**Instructor: Engr. Ma. Rizette Sayo**
6. Output	

```
// C++ code to illustrate queue in Standard Template Library (STL)
#include <iostream>
#include <queue>
using namespace std;
void display(queue <string> q)
{
queue <string> c = q;
while (!c.empty())
{
cout << " " << c.front();
c.pop();
}
cout << "\n";
}
int main()
{
queue <string> a;
a.push("Kenn");
a.push("Don");
a.push("Leoj");
cout << "The queue a is :";
display(a);
cout << "a.empty() :" << a.empty() << "\n";
cout << "a.size() : " << a.size() << "\n";
cout << "a.front() : " << a.front() << "\n";
cout << "a.back() : " << a.back() << "\n";
cout << "a.pop() : ";
a.pop();
display(a);

a.push("Tan");
cout << "The queue a is :";
display(a);
return 0;
}
```

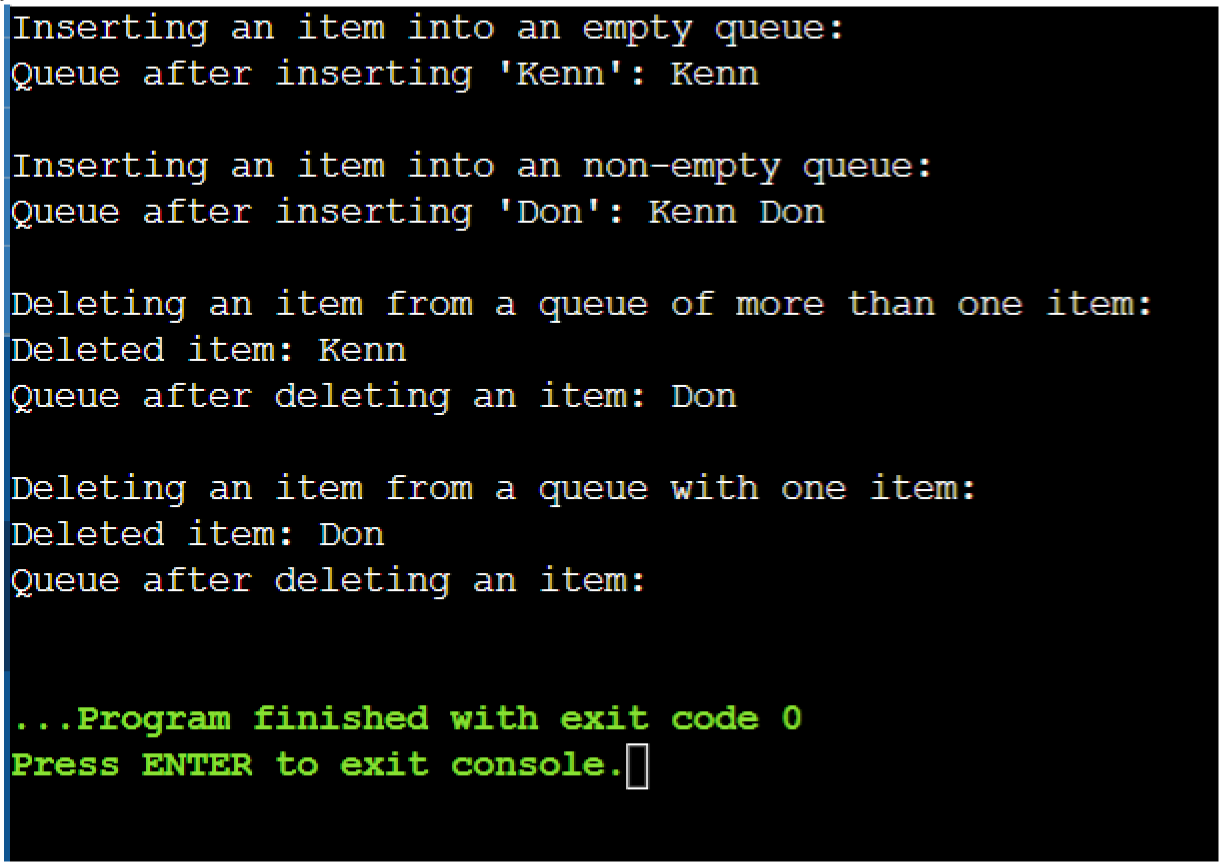


```
The queue a is : Kenn Don Leoj
a.empty() :0
a.size() : 3
a.front() : Kenn
a.back() : Leoj
a.pop() :  Don Leoj
The queue a is : Don Leoj Tan


...Program finished with exit code 0
Press ENTER to exit console.
```

Observation: The first operation displayed the elements inside the queue
then the operator looked if its empty then it display 0 as false
then the operation looked whats in the front of the queue and displayed it
then the back operation is looked whats in the back of the queue and displayed it
after pop() the front of the queue then it displays the remaining elements
before displaying the final queue elements, it push a new element which is tan

Table 5-1. Queues using C++ STL

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

using namespace std;


struct Node {
   string data;
   Node* next;
};


class Queue {
private:
   Node* front;
   Node* rear;

public:

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


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


    void enqueue(string item) {
        Node* newNode = new Node();
        newNode->data = item;
        newNode->next = nullptr;

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


    string dequeue() {
        if (isEmpty()) {
            return "Queue is empty";
        } else {
            string item = front->data;
            Node* temp = front;
            front = front->next;

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

            delete temp;
            return item;
        }
    }

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

int main() {
    Queue q;

    q.enqueue("Kenn");
    cout << "Inserting an item into an empty queue:"<<endl;
    cout << "Queue after inserting 'Kenn': ";
    q.display();

    q.enqueue("Don");
    cout << "\nInserting an item into an non-empty queue:"<<endl;
    cout << "Queue after inserting 'Don': ";
    q.display();


    cout << "\nDeleting an item from a queue of more than one item:"<<endl;
    cout << "Deleted item: " << q.dequeue() << endl;
    cout << "Queue after deleting an item: ";
    q.display();



    cout << " \nDeleting an item from a queue with one item:"<<endl;
    cout << "Deleted item: " << q.dequeue() << endl;
    cout << "Queue after deleting an item: ";
    q.display();


    return 0;
}
```

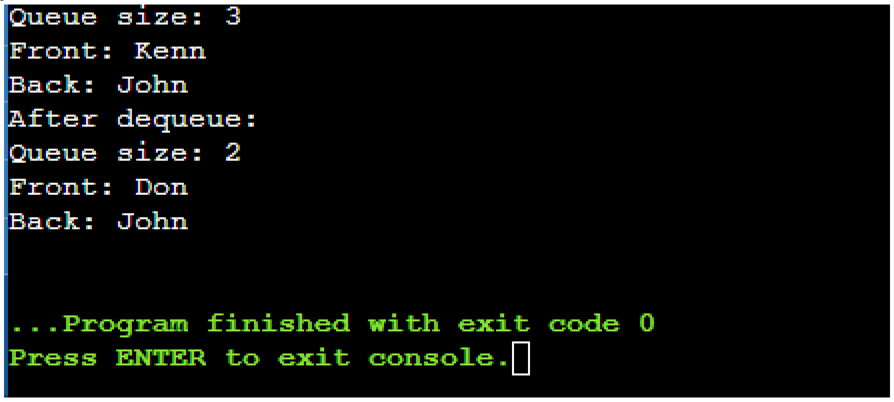


```
Inserting an item into an empty queue:
Queue after inserting 'Kenn': Kenn

Inserting an item into an non-empty queue:
Queue after inserting 'Don': Kenn Don

Deleting an item from a queue of more than one item:
Deleted item: Kenn
Queue after deleting an item: Don

Deleting an item from a queue with one item:
Deleted item: Don
Queue after deleting an item:


...Program finished with exit code 0
Press ENTER to exit console.
```

Table 5-2. Queues using Linked List Implementation

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

using namespace std;

class Queue {
private:
    string* q_array;
    int q_capacity;
    int q_size;
    int q_front;
    int q_back;
public:
        Queue(int capacity) {
        q_array = new string[capacity];
        q_capacity = capacity;
        q_size = 0;
        q_front = 0;
        q_back = 0;
    }

       ~Queue() {
        delete[] q_array;
    }

       bool isEmpty() {
        return (q_size == 0);
    }

    // Size Return size
    int size() {
        return q_size;
    }

       void clear() {
        q_size = 0;
        q_front = 0;
        q_back = 0;
    }

        string front() {
        if (isEmpty()) {
            throw std::runtime_error("Queue is empty");
        }
        return q_array[q_front];
    }


    string back() {
        if (isEmpty()) {
            throw std::runtime_error("Queue is empty");
        }
        return q_array[(q_back - 1 + q_capacity) % q_capacity];
    }


    void enqueue(string item) {
        if (q_size == q_capacity) {
            throw std::runtime_error("Queue is full");
        }
        q_array[q_back] = item;
        q_back = (q_back + 1) % q_capacity;
        q_size++;
    }


    string dequeue() {
        if (isEmpty()) {
            throw std::runtime_error("Queue is empty");
        }
        string item = q_array[q_front];
        q_front = (q_front + 1) % q_capacity;
        q_size--;
        return item;
    }

        Queue(const Queue& other) {
        q_capacity = other.q_capacity;
        q_size = other.q_size;
        q_front = other.q_front;
        q_back = other.q_back;
        q_array = new string[q_capacity];
        for (int i = 0; i < q_capacity; i++) {
            q_array[i] = other.q_array[i];
        }
    }

       Queue& operator=(const Queue& other) {
        if (this != &other) {
            delete[] q_array;
            q_capacity = other.q_capacity;
            q_size = other.q_size;
            q_front = other.q_front;
            q_back = other.q_back;
            q_array = new string[q_capacity];
            for (int i = 0; i < q_capacity; i++) {
                q_array[i] = other.q_array[i];
            }
        }
        return *this;
    }
};
```

```
int main() {
    Queue q(5);
    q.enqueue("Kenn");
    q.enqueue("Don");
    q.enqueue("John");

    cout << "Queue size: " << q.size() << endl;
    cout << "Front: " << q.front() << endl;
    cout << "Back: " << q.back() << endl;

    q.dequeue();
    cout << "After dequeue: " << endl;
    cout << "Queue size: " << q.size() << endl;
    cout << "Front: " << q.front() << endl;
    cout << "Back: " << q.back() << endl;

}
```

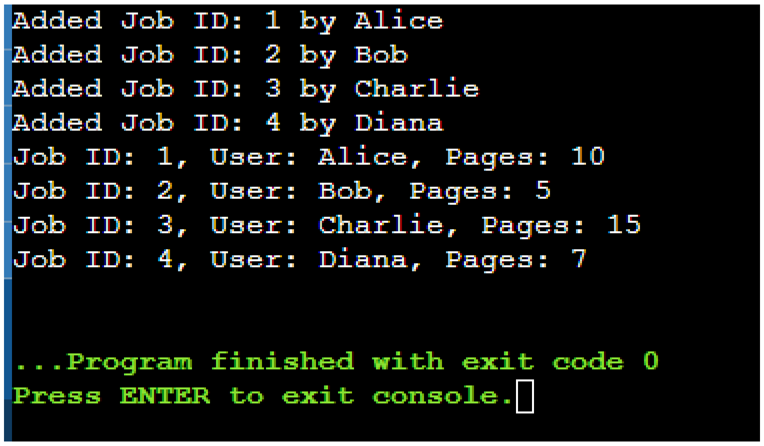


```
Queue size: 3
Front: Kenn
Back: John
After dequeue:
Queue size: 2
Front: Don
Back: John


...Program finished with exit code 0
Press ENTER to exit console.
```

Table 5-3. Queues using Array Implementation

7. Supplementary Activity

```
#include <iostream>
#include <string>
using namespace std;


class Job {
public:
    int jobID;
    string userName;
    int numPages;


    Job(int id, string user, int pages) {
        jobID = id;
        userName = user;
        numPages = pages;
    }
    void displayJob() {
        cout << "Job ID: " << jobID << ", User: " << userName << ", Pages: " << numPages << endl;
    }
};


class Node {
public:
    Job* job;
    Node* next;


    Node(Job* newJob) {
        job = newJob;
        next = nullptr;
    }
};


class Printer {
private:
    Node* front;
    Node* rear;

public:
    Printer() {
        front = nullptr;
        rear = nullptr;
    }


    void addJob(int id, string user, int pages) {
        Job* newJob = new Job(id, user, pages);
        Node* newNode = new Node(newJob);

        if (rear == nullptr) {
            front = rear = newNode;
        } else {
            rear->next = newNode;
            rear = newNode;
        }
        cout << "Added Job ID: " << id << " by " << user << endl;
    }


    void processJobs() {
        while (front != nullptr) {
            Node* temp = front;
            front->job->displayJob();
            front = front->next;
            delete temp;
        }
        rear = nullptr;
    }
};


int main() {
    Printer printer;


    printer.addJob(1, "Alice", 10);
    printer.addJob(2, "Bob", 5);
    printer.addJob(3, "Charlie", 15);
    printer.addJob(4, "Diana", 7);



    printer.processJobs();

    return 0;
}
```



```
Added Job ID: 1 by Alice
Added Job ID: 2 by Bob
Added Job ID: 3 by Charlie
Added Job ID: 4 by Diana
Job ID: 1, User: Alice, Pages: 10
Job ID: 2, User: Bob, Pages: 5
Job ID: 3, User: Charlie, Pages: 15
Job ID: 4, User: Diana, Pages: 7


...Program finished with exit code 0
Press ENTER to exit console.
```

A linked list works well here because we don't know how many print jobs will be sent to the printer ahead of time. Since new jobs keep getting added, a linked list allows us to grow the list dynamically without worrying about setting a fixed size, like we would have to with an array.

Plus, adding or removing jobs is simpler in a linked list. We don't need to worry about shifting elements or resizing, which makes it a more flexible choice for handling the print queue. This way, jobs can be processed in the order they were added, and we won't run into any space issues.

The output reflects a First-In, First-Out (FIFO) approach where jobs are processed in the same order they were added to the queue. Each job is printed in sequence, from Alice's to Diana's, without skipping or reordering. This happens because the linked list structure ensures that the first job added is always the first one processed, mimicking how a real shared printer handles tasks.

8. Conclusion

Through this activity, I learned how queues work and the importance of the First-In, First-Out (FIFO) method in organizing data. By using C++ with STL, linked lists, and arrays, I saw how each implementation affects performance and flexibility. The print queue simulation was a great example of how linked lists can handle unpredictable data sizes more efficiently

than arrays. I felt I did well in coding and understanding the basic operations, but I could improve by exploring the time complexity and efficiency differences between the implementations. Overall, this activity helped me grasp the real-world use of queues in managing tasks.
9. Assessment Rubric