Hands-on Activity 5.1	
Queues	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 09 - 11 - 25
Section: CPE21S4	**Date Submitted:** 09 - 11 - 25
Name(s): Santiago, David Owen A.	**Instructor:** Engr. Jimlord Quejado

6. Output

Table 5.1 Queues using C++ STL

Code:

```cpp
#include <iostream>
#include <queue>

void printQueue(std::queue<std::string> q) {
   while (!q.empty()) {
       std::cout << q.front() << " ";
       q.pop();
   }std::cout<<'\n';
}
int main() {

   std::queue<std::string> queue;
   std::string students[4] = {"Roman", "Jimlord", "Jihan", "Robin"};
   std::cout << "Students Array: ";

   for (int i = 0; i < 4; i++) {
       std::cout << students[i] << " ";
       queue.push(students[i]);
   }std::cout << '\n';

   std::cout << "The queue: ";
   printQueue(queue);
   std::cout << "Front of the queue: " << queue.front() << '\n';
   std::cout << "Back of the queue: " << queue.back() << '\n';
   return 0;
}
```

Output:

```
"D:\College\Year 2 CPE\Sem 1\CPE 010\Queue\cmake-build-debug\untitled1.exe"
Students Array: Roman Jimlord Jihan Robin
The queue: Roman Jimlord Jihan Robin
Front of the queue: Roman
Back of the queue: Robin


Process finished with exit code 0
```

In the queue, each student is passed in proper order with the first element being the first member of the array. This is because the following elements are added at the end of the queue, hence, the order is consistent. I then created a void function that uses a while loop to print out each element of the queue. I also verified the front and back of the queue which are the first and last elements added respectively.

Table 5.2 Queues using Linked List Implementation

Code:

Header File:

```
//Created on 09-09-25
//by David Owen Santiago, CPE21S4

#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;
      }
};

template<typename T>
class Queue{
   private:
      Node<T> *front, *rear;
      int size = 0;

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

      // Copy constructor
      Queue(const Queue<T>& original) {
            front = rear = nullptr;
            size = 0;

            Node<T>* temp = original.front;
            while (temp != nullptr) {
               enqueue(temp->data);
               temp = temp->next;
            }
            std::cout << "Queue copied successfully.\n";
         }
      //to deallocate memory (destructor)
      ~Queue(){
            clear();
         }
```

```
      //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 << "Enqueued to an empty queue: " << new_data <<'\n';
            size++;
            return;
         }
         rear->next = new_node;
         rear = new_node;
         std::cout << "Enqueued: " << new_data <<'\n';
      }

      //dequeue
      void dequeue(){
         //check if empty
         if (isEmpty()){
            std::cout << "Queue is empty.\n";
            return;
         }
         //create temporary node to store original front
         Node<T> *temp = front;

         //if front points to empty, then the queue is empty, therefore rear also points
to null
         if (front == nullptr) {
            rear = nullptr;
         }

         else{
            //reassign front to the data next to the og front node
            front = front->next;
         }
         delete temp;
         std::cout << "Successfully dequeued.\n";
         size--;
      }
      //getFront
      void getFront(){
         if (isEmpty()){
            std::cout << "Queue is empty.\n";
            return;
         }
         std::cout << "Front is: "<< front->data << '\n';
      }

      //getRear
      void getRear(){
         if (isEmpty()){
            std::cout << "Queue is empty.\n";
            return;
         }
         std::cout << "Rear is: "<< rear->data << '\n';
      }

      //display
      void printQueue(){
```

```
        if (isEmpty()){
            std::cout << "Queue is empty.\n";
            return;
        }
        Node<T> *temp = front;
        std::cout << "Current Queue: ";
        while (temp != nullptr){
            std::cout << temp->data << ' ';
            temp = temp->next;
        } std::cout<<'\n';
    }

    int QueueSize() {
        return size;
    }

    void clear() {
        while (!isEmpty()) {
            dequeue();
        }
    }

};
#endif
```

Main:

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

void line() {
   std::cout << "----------------\n";
}
int main() {

   Queue<int> q;

   if(q.isEmpty()) {
       std::cout << "Queue is empty\n";
   }
   line();

   for (int i = 0; i < 5; i++) {
       q.enqueue(i + 1);
   }
   q.printQueue();
   q.getFront();
   q.getRear();

   line();

   std::cout << "Copying Queue.\n";
   Queue<int> q2(q);
   q2.printQueue();
   q2.getFront();
   q2.getRear();

   line();

   q.clear();
```

```
    q.printQueue();
    q.getFront();
    q.getRear();

    line();

    return 0;
}
```

Output:

```
"D:\College\Year 2 CPE\Sem 1\CPE 010\Queue\cmake-build-debug\untitled1.exe"
A queue has been created.
Queue is empty
---------------
Enqueued to an empty queue: 1
Enqueued: 2
Enqueued: 3
Enqueued: 4
Enqueued: 5
Current Queue: 1 2 3 4 5
Front is: 1
Rear is: 5
---------------
Copying Queue.
Enqueued to an empty queue: 1
Enqueued: 2
Enqueued: 3
Enqueued: 4
Enqueued: 5
Queue copied successfully.
Current Queue: 1 2 3 4 5
Front is: 1
Rear is: 5
---------------
Successfully dequeued.
Successfully dequeued.
Successfully dequeued.
Successfully dequeued.
Successfully dequeued.
Queue is empty.
Queue is empty.
Queue is empty.
---------------
Successfully dequeued.
Successfully dequeued.
Successfully dequeued.
Successfully dequeued.
Successfully dequeued.

Process finished with exit code 0
```

In the linked list implementation of a queue, I created the classes, functions, and operations in a separate header file then included it in the main file. Using linked lists is much more efficient than using arrays when creating linked lists because not only is it dynamic but each node also has their own addresses that can be allocated and deallocated on the fly. This allows the front and rear locations to be consistent, making operations much easier and more efficient. Additionally, I included a copy constructor and destructor to complete the class queue.

Table 5.3. Queues using Array Implementation

Code:

Header File:

```
//Created on 09-11-25
//by David Owen Santiago, CPE21S4

#ifndef ARRAY_QUEUE_H
#define ARRAY_QUEUE_H
#include <iostream>

template<typename T>
class Queue {
private:
   T* arr;
   int capacity, front, rear, size;

public:
   // Constructor
   Queue(int max_size = 100) {
       capacity = max_size;
       arr = new T[capacity];
       front = 0;
       rear = -1;
       size = 0;
       std::cout << "Queue created.\n";
   }

   // Copy constructor
   Queue(const Queue<T>& orig) {
       std::cout << "Copying Queue.\n";
       capacity = orig.capacity;
       arr = new T[capacity];
       front = orig.front;
       rear = orig.rear;
       size = orig.size;

       for (int i = 0; i < size; ++i) {
           int index = front + i;
           if (index >= capacity) index -= capacity;
           arr[index] = orig.arr[index];
           std::cout << "Enqueued: " << arr[index] << '\n';
       }
       std::cout << "Queue copied.\n";
   }

   // Assignment Operator
   Queue<T>& operator=(const Queue<T>& orig) {
```

```
        std::cout << "Copying Queue.\n";
        capacity = orig.capacity;
        arr = new T[capacity];
        front = orig.front;
        rear = orig.rear;
        size = orig.size;
        for (int i = 0; i < size; ++i) {
            int index = front + i;
            if (index >= capacity) index -= capacity;
            arr[index] = orig.arr[index];
            std::cout << "Enqueued: " << arr[index] << '\n';
        }
        std::cout << "Queue assigned.\n";
        return *this;
    }

    // Destructor
    ~Queue() {
        delete[] arr;
        std::cout << "Queue destroyed.\n";
    }

    // isEmpty
    bool isEmpty() const {
        return size == 0;
    }

    // isFull
    bool isFull() const {
        return size == capacity;
    }

    // enqueue
    void enqueue(T value) {
        if (isFull()) {
            std::cout << "Queue is full.\n";
            return;
        }
        rear++;
        if (rear == capacity) rear = 0;
        arr[rear] = value;
        size++;
        std::cout << "Enqueued: " << value << '\n';
    }

    // dequeue
    void dequeue() {
        if (isEmpty()) {
            std::cout << "Queue is empty.\n";
            return;
        }
        std::cout << "Dequeued: " << arr[front] << '\n';
        front++;
        if (front == capacity) front = 0;
        size--;
    }

    // getFront
    void getFront() const {
        if (isEmpty()) {
            std::cout << "Queue is empty.\n";
            return;
```

```
        }
        std::cout << "Front is: " << arr[front] << '\n';
    }

    // getRear
    void getRear() const {
        if (isEmpty()) {
            std::cout << "Queue is empty.\n";
            return;
        }
        std::cout << "Rear is: " << arr[rear] << '\n';
    }

    // printQueue
    void printQueue() const {
        if (isEmpty()) {
            std::cout << "Queue is empty.\n";
            return;
        }
        std::cout << "Current Queue: ";
        for (int i = 0; i < size; ++i) {
            int index = front + i;
            if (index >= capacity) index -= capacity;
            std::cout << arr[index] << ' ';
        }
        std::cout << '\n';
    }

    // QueueSize
    int QueueSize() const {
        return size;
    }

    // clear
    void clear() {
        front = 0;
        rear = -1;
        size = 0;
        std::cout << "Queue cleared.\n";
    }
};

#endif
```

Main File:

```
#include <iostream>
//#include "queue.h"
#include "array_queue.h"

void line() {
    std::cout << "----------------\n";
}
int main() {

    Queue<int> q;
    for (int i = 1; i <= 5; i++) {
        q.enqueue(i);
    }
    q.printQueue();
```

```
    line();

    Queue<int> q2(q);

    q2.printQueue();
    line();

    Queue<int> q3 = q;
    line();

    return 0;
}
```

Output:

```
"D:\College\Year 2 CPE\Sem 1\CPE 010\Queue\cmake-build-debug\untitled1.exe"
Queue created.
Enqueued: 1
Enqueued: 2
Enqueued: 3
Enqueued: 4
Enqueued: 5
Current Queue: 1 2 3 4 5
----------------
Copying Queue.
Enqueued: 1
Enqueued: 2
Enqueued: 3
Enqueued: 4
Enqueued: 5
Queue copied.
Current Queue: 1 2 3 4 5
----------------
Copying Queue.
Enqueued: 1
Enqueued: 2
Enqueued: 3
Enqueued: 4
Enqueued: 5
Queue copied.
----------------
Queue destroyed.
Queue destroyed.
Queue destroyed.

Process finished with exit code 0
```

The array implementation of the queue uses a circular queue that logically connects the rear element toward the front element to ensure that they use the same array space through controlled indexing. In this implementation, the front and rear are cycled through to avoid wasting array space.

7. Supplementary Activity

Problem Title: Shared Printer Simulation using Queues
Problem Definition: In this activity, we'll simulate a queue for a shared printer in an office. In any corporate office, usually, the printer is shared across the whole floor in the printer room. All the computers in this room are connected to the same printer. But a printer can do only one printing job at any point in time, and it also takes some time to complete any job. In the meantime, some other user can send another print request. In such a case, a printer needs to store all the pending jobs somewhere so that it can take them up once its current task is done.

Perform the following steps to solve the activity. Make sure that you include a screenshot of the source code for each.
From the get-go: You must choose whether you are making a linked list or array implementation.
You may NOT use the STL.

1. Create a class called Job (comprising an ID for the job, the name of the user who submitted it, and the number of pages).

2. Create a class called Printer. This will provide an interface to add new jobs and process all the jobs added so far.

3. To implement the printer class, it will need to store all the pending jobs. We'll implement a very basic strategy – first come, first served. Whoever submits the job first will be the first to get the job done.

4. Finally, simulate a scenario where multiple people are adding jobs to the printer, and the printer is processing them one by one.

5. Defend your choice of internal representation: Why did you use arrays or linked lists?

Header File:

```
//Created on 09-09-25
//by David Owen Santiago, CPE21S4
#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;
   }
};

template<typename T>
class Queue {
```

```
private:
    Node<T>* front;
    Node<T>* rear;
    int size = 0;

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

    Queue(const Queue<T>& original) {
        front = rear = nullptr;
        size = 0;
        Node<T>* temp = original.front;
        while (temp != nullptr) {
            enqueue(temp->data);
            temp = temp->next;
        }
        std::cout << "Queue copied successfully.\n";
    }

    ~Queue() {
        clear();
    }

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

    void enqueue(T new_data) {
        Node<T>* new_node = new Node<T>(new_data);
        if (isEmpty()) {
            front = rear = new_node;
            std::cout << "Enqueued to an empty queue\n";
            size++;
            return;
        }
        rear->next = new_node;
        rear = new_node;
        std::cout << "Enqueued\n";
        size++;
    }

    void dequeue() {
        if (isEmpty()) {
            std::cout << "Queue is empty.\n";
            return;
        }
        Node<T>* temp = front;
        front = front->next;
        if (front == nullptr) rear = nullptr;
        delete temp;
        std::cout << "Successfully dequeued.\n";
        size--;
    }

    void getFront() {
        if (isEmpty()) {
            std::cout << "Queue is empty.\n";
            return;
        }
        std::cout << "Front is: " << front->data << '\n';
```

```
    }

    void getRear() {
        if (isEmpty()) {
            std::cout << "Queue is empty.\n";
            return;
        }
        std::cout << "Rear is: " << rear->data << '\n';
    }

    void printQueue() {
        if (isEmpty()) {
            std::cout << "Queue is empty.\n";
            return;
        }
        Node<T>* temp = front;
        std::cout << "Current Queue: ";
        while (temp != nullptr) {
            std::cout << temp->data << ' ';
            temp = temp->next;
        }
        std::cout << '\n';
    }

    int QueueSize() {
        return size;
    }

    void clear() {
        while (!isEmpty()) {
            dequeue();
        }
    }
};

#endif
```

Main function:

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

class Job {
public:
    int jobID;
    std::string userName;
    int pageCount;

    Job(int id, std::string user, int pages) {
        jobID = id;
        userName = user;
        pageCount = pages;
    }
    Job() : jobID(0), userName(""), pageCount(0) {}

    void printDeets() const {
        std::cout << "Job ID: " << jobID << "\n";
        std::cout << "User Name: " << userName << "\n";
        std::cout << "Page Count: " << pageCount << "\n";
    }
};
```

```
// Overload << for Job
std::ostream& operator<<(std::ostream& os, const Job& job) {
   os << "Job ID: " << job.jobID
      << ", User Name: " << job.userName
      << ", Page Count: " << job.pageCount;
   return os;
}

class Printer {
private:
   Queue<Job> jobQueue;

public:
   void addJob(const Job& newJob) {
       jobQueue.enqueue(newJob);
       std::cout << "Added job to queue\n";
       newJob.printDeets();
   }

   void processJobs() {
       while (!jobQueue.isEmpty()) {
           std::cout << "Processing job...\n";
           jobQueue.getFront();
           jobQueue.dequeue();
       }
       std::cout << "All jobs processed.\n";
   }
};

int main() {
   Printer companyPrinter;
   companyPrinter.addJob(Job(241, "Roman", 100));
   companyPrinter.addJob(Job(242, "Jimlord", 200));
   companyPrinter.addJob(Job(243, "Robin", 300));

   std::cout << "\nProcessing Jobs \n";
   companyPrinter.processJobs();

   return 0;
}
```

Output:

```
"D:\College\Year 2 CPE\Sem 1\CPE 010\Queue\cmake-build-debug\untitled1.exe"
A queue has been created.
Enqueued to an empty queue
Added job to queue
Job ID: 241
User Name: Roman
Page Count: 100
Enqueued
Added job to queue
Job ID: 242
User Name: Jimlord
Page Count: 200
Enqueued
Added job to queue
Job ID: 243
User Name: Robin
Page Count: 300

Processing Jobs
Processing job...
Front is: Job ID: 241, User Name: Roman, Page Count: 100
Successfully dequeued.
Processing job...
Front is: Job ID: 242, User Name: Jimlord, Page Count: 200
Successfully dequeued.
Processing job...
Front is: Job ID: 243, User Name: Robin, Page Count: 300
Successfully dequeued.
All jobs processed.

Process finished with exit code 0
```

8. Conclusion

In my linked list implementation of a queue, I organized all the classes, functions, and operations inside a separate header file, which I then included in the main file to keep it modular. I chose to use linked lists instead of arrays because they're more flexible and dynamic where each node has its own memory address, which can be allocated and freed as needed. I am also much more confident in using linked lists compared to arrays when making queues since I understand the sorting much better than the array indexing. Moreover, this makes it easier to manage the front and rear of the queue without worrying about fixed sizes or shifting elements. I also kept the job classes in the main file to keep the queue header strictly for the queue operations. It also keeps the operations consistent and efficient. To complete the class, I added a copy constructor and a destructor to handle memory properly.

9. Assessment Rubric