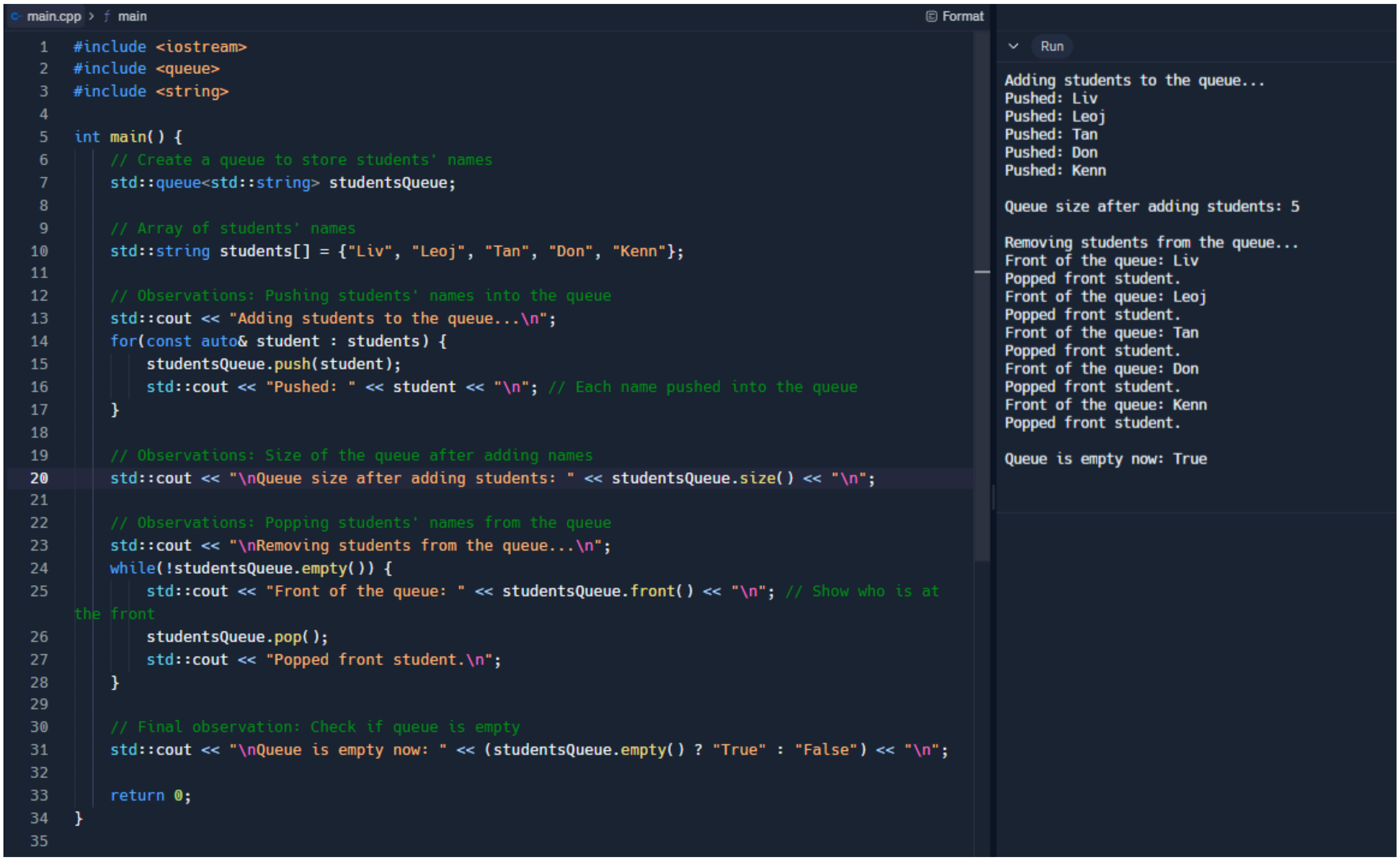
Activity No. 5

QUEUES

Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 07/10/24
Section: CPE21S4	**Date Submitted:** 07/10/24
Name(s): Prince Wally G. Esteban	**Instructor:** Mrs. Marizette Sayo

6. Output

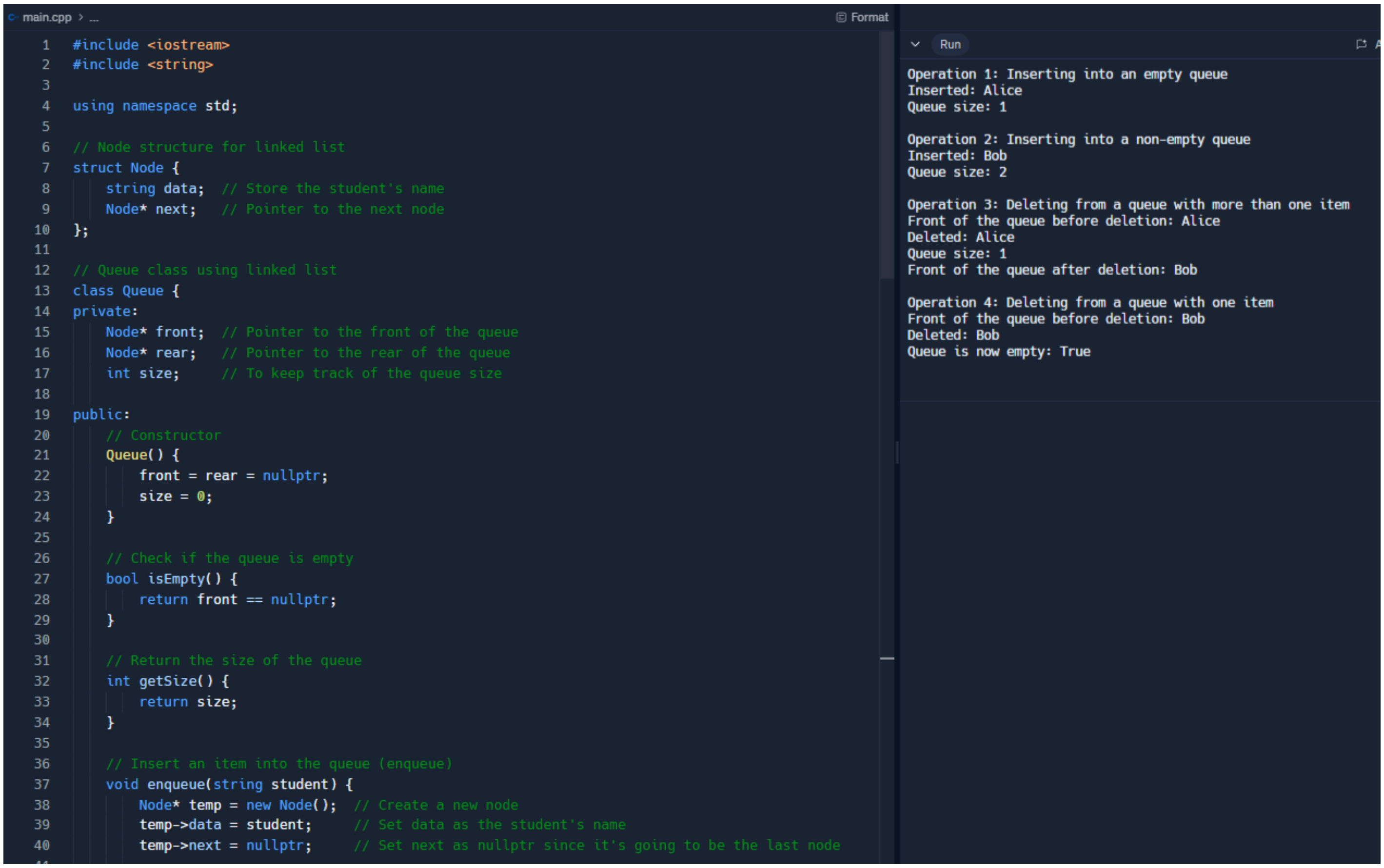
TABLE 5.1



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

int main() {
    // Create a queue to store students' names
    std::queue<std::string> studentsQueue;

    // Array of students' names
    std::string students[] = {"Liv", "Leoj", "Tan", "Don", "Kenn"};

    // Observations: Pushing students' names into the queue
    std::cout << "Adding students to the queue...\n";
    for(const auto& student : students) {
        studentsQueue.push(student);
        std::cout << "Pushed: " << student << "\n"; // Each name pushed into the queue
    }

    // Observations: Size of the queue after adding names
    std::cout << "\nQueue size after adding students: " << studentsQueue.size() << "\n";

    // Observations: Popping students' names from the queue
    std::cout << "\nRemoving students from the queue...\n";
    while(!studentsQueue.empty()) {
        std::cout << "Front of the queue: " << studentsQueue.front() << "\n"; // Show who is at the front
        studentsQueue.pop();
        std::cout << "Popped front student.\n";
    }

    // Final observation: Check if queue is empty
    std::cout << "\nQueue is empty now: " << (studentsQueue.empty() ? "True" : "False") << "\n";

    return 0;
}
```

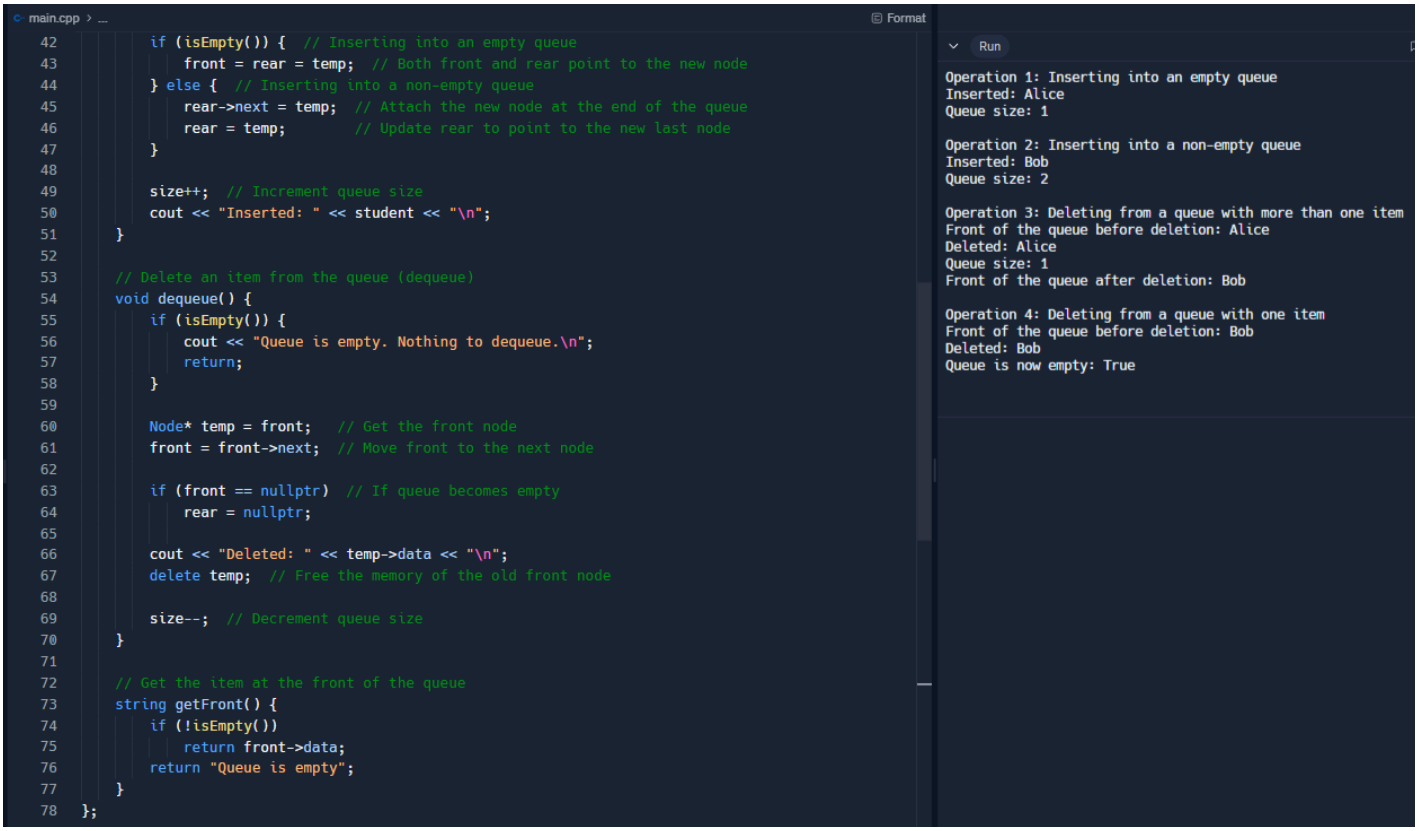
```
Adding students to the queue...
Pushed: Liv
Pushed: Leoj
Pushed: Tan
Pushed: Don
Pushed: Kenn

Queue size after adding students: 5

Removing students from the queue...
Front of the queue: Liv
Popped front student.
Front of the queue: Leoj
Popped front student.
Front of the queue: Tan
Popped front student.
Front of the queue: Don
Popped front student.
Front of the queue: Kenn
Popped front student.

Queue is empty now: True
```

TABLE 5.2



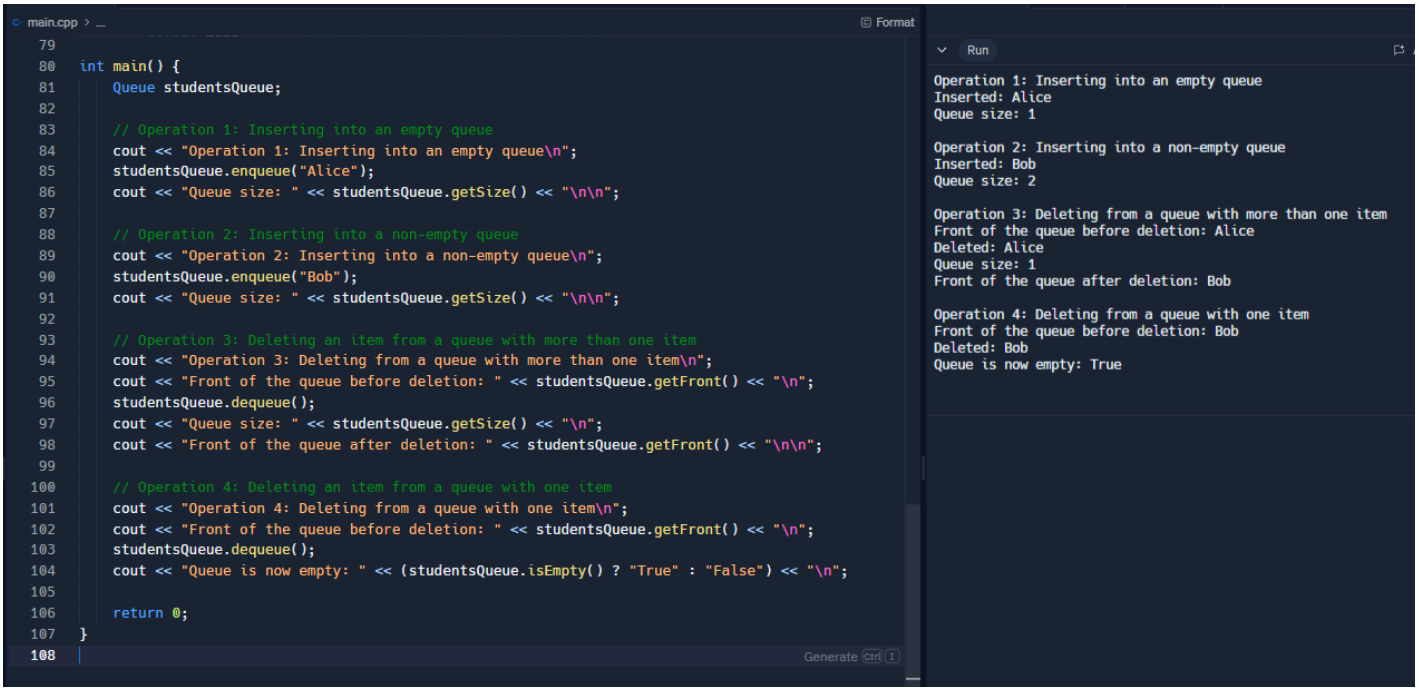
```cpp
#include <iostream>
#include <string>

using namespace std;

// Node structure for linked list
struct Node {
    string data;  // Store the student's name
    Node* next;   // Pointer to the next node
};

// Queue class using linked list
class Queue {
private:
    Node* front;  // Pointer to the front of the queue
    Node* rear;   // Pointer to the rear of the queue
    int size;     // To keep track of the queue size

public:
    // Constructor
    Queue() {
        front = rear = nullptr;
        size = 0;
    }

    // Check if the queue is empty
    bool isEmpty() {
        return front == nullptr;
    }

    // Return the size of the queue
    int getSize() {
        return size;
    }

    // Insert an item into the queue (enqueue)
    void enqueue(string student) {
        Node* temp = new Node();  // Create a new node
        temp->data = student;     // Set data as the student's name
        temp->next = nullptr;     // Set next as nullptr since it's going to be the last node
```

```
Operation 1: Inserting into an empty queue
Inserted: Alice
Queue size: 1

Operation 2: Inserting into a non-empty queue
Inserted: Bob
Queue size: 2

Operation 3: Deleting from a queue with more than one item
Front of the queue before deletion: Alice
Deleted: Alice
Queue size: 1
Front of the queue after deletion: Bob

Operation 4: Deleting from a queue with one item
Front of the queue before deletion: Bob
Deleted: Bob
Queue is now empty: True
```



```cpp
        if (isEmpty()) {  // Inserting into an empty queue
            front = rear = temp;  // Both front and rear point to the new node
        } else {  // Inserting into a non-empty queue
            rear->next = temp;  // Attach the new node at the end of the queue
            rear = temp;        // Update rear to point to the new last node
        }

        size++;  // Increment queue size
        cout << "Inserted: " << student << "\n";
    }

    // Delete an item from the queue (dequeue)
    void dequeue() {
        if (isEmpty()) {
            cout << "Queue is empty. Nothing to dequeue.\n";
            return;
        }

        Node* temp = front;   // Get the front node
        front = front->next;  // Move front to the next node

        if (front == nullptr)  // If queue becomes empty
            rear = nullptr;

        cout << "Deleted: " << temp->data << "\n";
        delete temp;  // Free the memory of the old front node

        size--;  // Decrement queue size
    }

    // Get the item at the front of the queue
    string getFront() {
        if (!isEmpty())
            return front->data;
        return "Queue is empty";
    }
};
```

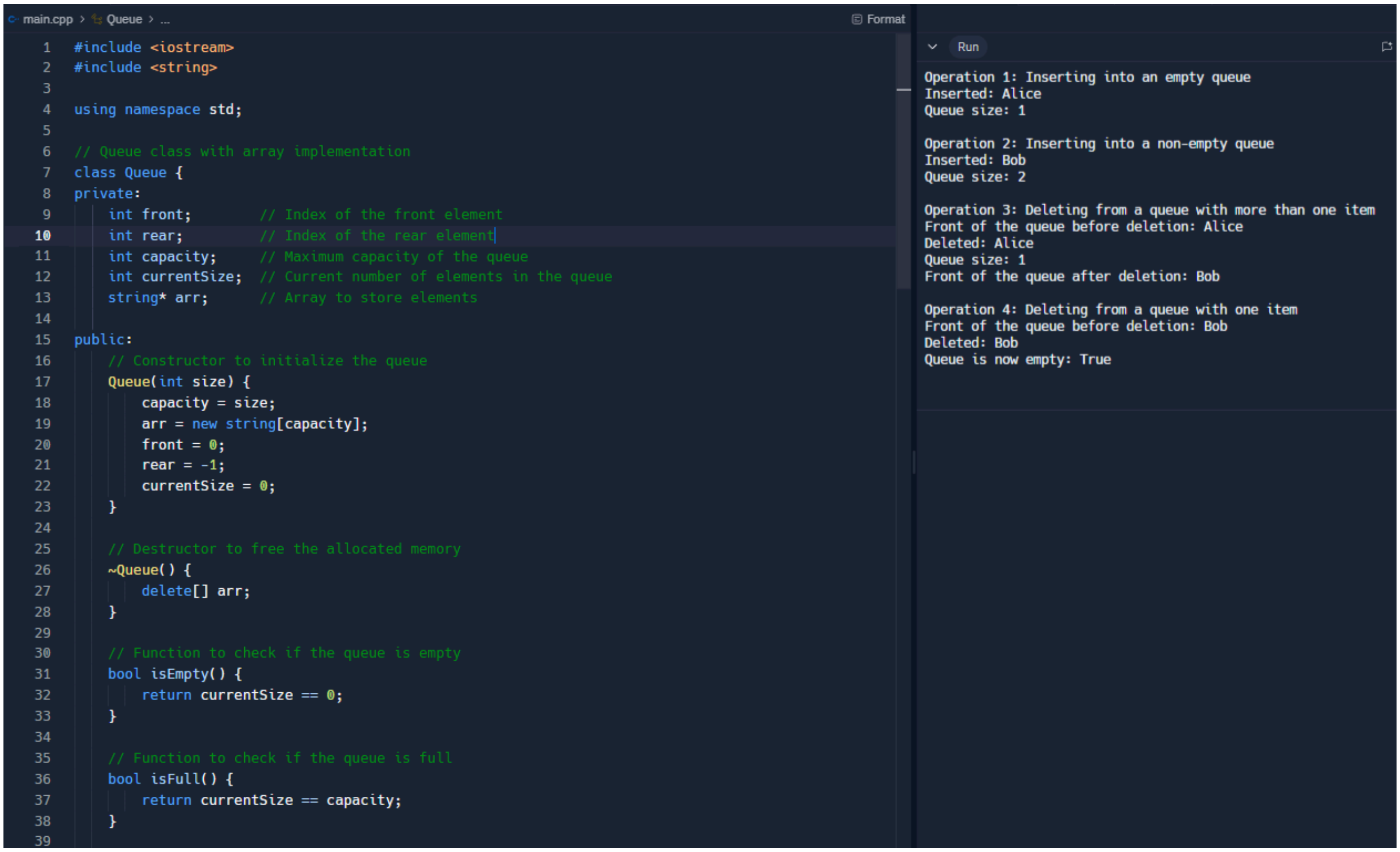
```
Operation 1: Inserting into an empty queue
Inserted: Alice
Queue size: 1

Operation 2: Inserting into a non-empty queue
Inserted: Bob
Queue size: 2

Operation 3: Deleting from a queue with more than one item
Front of the queue before deletion: Alice
Deleted: Alice
Queue size: 1
Front of the queue after deletion: Bob

Operation 4: Deleting from a queue with one item
Front of the queue before deletion: Bob
Deleted: Bob
Queue is now empty: True
```



```cpp

int main() {
    Queue studentsQueue;

    // Operation 1: Inserting into an empty queue
    cout << "Operation 1: Inserting into an empty queue\n";
    studentsQueue.enqueue("Alice");
    cout << "Queue size: " << studentsQueue.getSize() << "\n\n";

    // Operation 2: Inserting into a non-empty queue
    cout << "Operation 2: Inserting into a non-empty queue\n";
    studentsQueue.enqueue("Bob");
    cout << "Queue size: " << studentsQueue.getSize() << "\n\n";

    // Operation 3: Deleting an item from a queue with more than one item
    cout << "Operation 3: Deleting from a queue with more than one item\n";
    cout << "Front of the queue before deletion: " << studentsQueue.getFront() << "\n";
    studentsQueue.dequeue();
    cout << "Queue size: " << studentsQueue.getSize() << "\n";
    cout << "Front of the queue after deletion: " << studentsQueue.getFront() << "\n\n";

    // Operation 4: Deleting an item from a queue with one item
    cout << "Operation 4: Deleting from a queue with one item\n";
    cout << "Front of the queue before deletion: " << studentsQueue.getFront() << "\n";
    studentsQueue.dequeue();
    cout << "Queue is now empty: " << (studentsQueue.isEmpty() ? "True" : "False") << "\n";

    return 0;
}

```

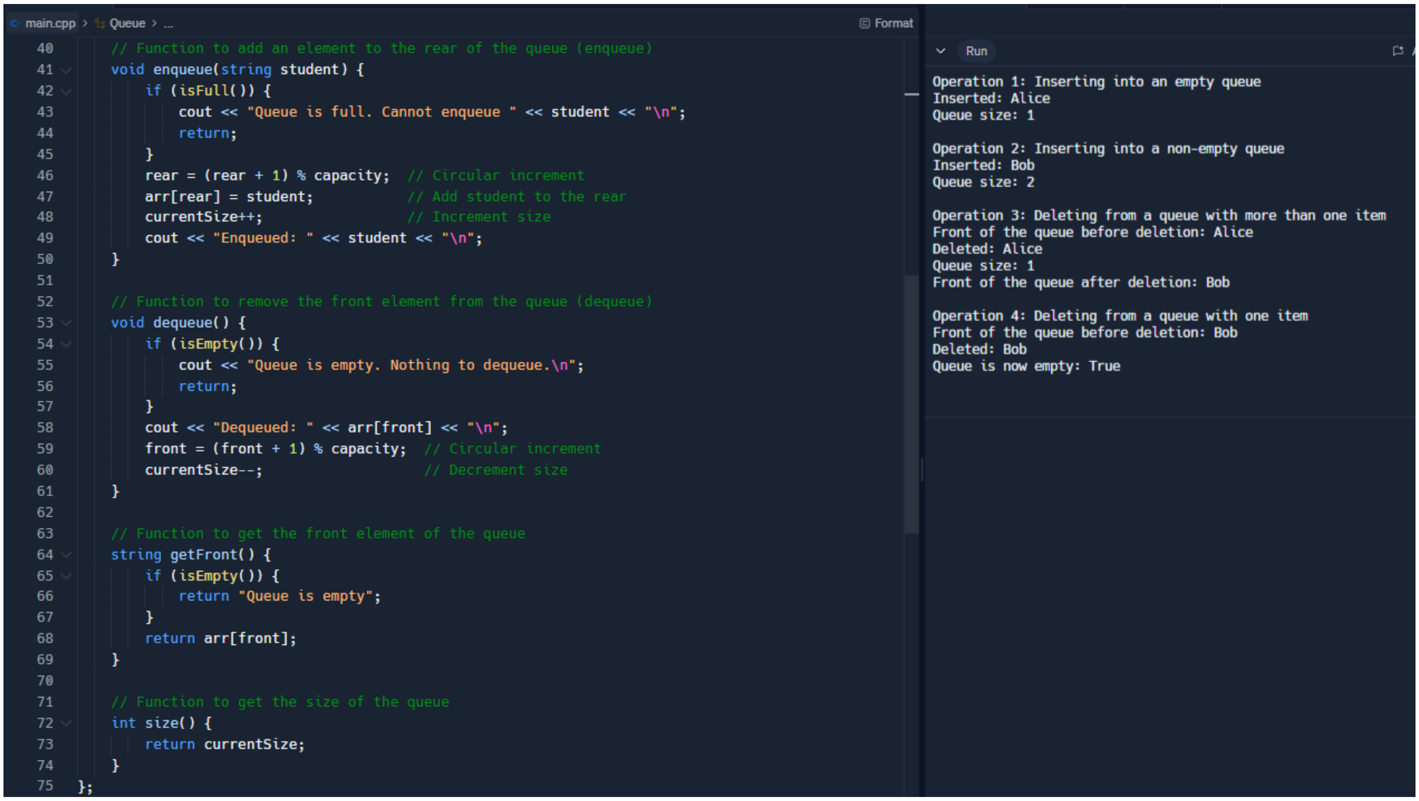
```
Operation 1: Inserting into an empty queue
Inserted: Alice
Queue size: 1

Operation 2: Inserting into a non-empty queue
Inserted: Bob
Queue size: 2

Operation 3: Deleting from a queue with more than one item
Front of the queue before deletion: Alice
Deleted: Alice
Queue size: 1
Front of the queue after deletion: Bob

Operation 4: Deleting from a queue with one item
Front of the queue before deletion: Bob
Deleted: Bob
Queue is now empty: True
```

TABLE 5.3



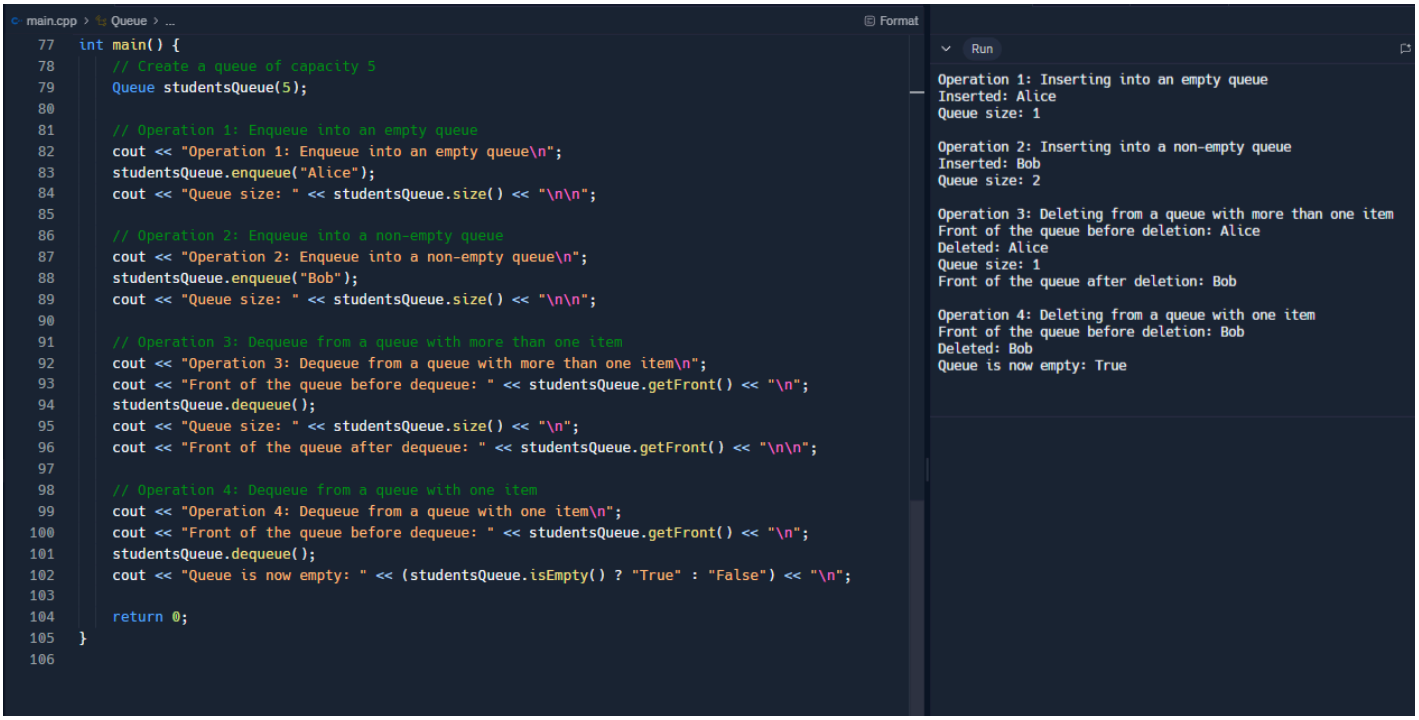
```cpp
#include <iostream>
#include <string>

using namespace std;

// Queue class with array implementation
class Queue {
private:
    int front;        // Index of the front element
    int rear;         // Index of the rear element
    int capacity;     // Maximum capacity of the queue
    int currentSize;  // Current number of elements in the queue
    string* arr;      // Array to store elements

public:
    // Constructor to initialize the queue
    Queue(int size) {
        capacity = size;
        arr = new string[capacity];
        front = 0;
        rear = -1;
        currentSize = 0;
    }

    // Destructor to free the allocated memory
    ~Queue() {
        delete[] arr;
    }

    // Function to check if the queue is empty
    bool isEmpty() {
        return currentSize == 0;
    }

    // Function to check if the queue is full
    bool isFull() {
        return currentSize == capacity;
    }

```

```
Operation 1: Inserting into an empty queue
Inserted: Alice
Queue size: 1

Operation 2: Inserting into a non-empty queue
Inserted: Bob
Queue size: 2

Operation 3: Deleting from a queue with more than one item
Front of the queue before deletion: Alice
Deleted: Alice
Queue size: 1
Front of the queue after deletion: Bob

Operation 4: Deleting from a queue with one item
Front of the queue before deletion: Bob
Deleted: Bob
Queue is now empty: True
```



```cpp
    // Function to add an element to the rear of the queue (enqueue)
    void enqueue(string student) {
        if (isFull()) {
            cout << "Queue is full. Cannot enqueue " << student << "\n";
            return;
        }
        rear = (rear + 1) % capacity;  // Circular increment
        arr[rear] = student;           // Add student to the rear
        currentSize++;                 // Increment size
        cout << "Enqueued: " << student << "\n";
    }

    // Function to remove the front element from the queue (dequeue)
    void dequeue() {
        if (isEmpty()) {
            cout << "Queue is empty. Nothing to dequeue.\n";
            return;
        }
        cout << "Dequeued: " << arr[front] << "\n";
        front = (front + 1) % capacity;  // Circular increment
        currentSize--;                   // Decrement size
    }

    // Function to get the front element of the queue
    string getFront() {
        if (isEmpty()) {
            return "Queue is empty";
        }
        return arr[front];
    }

    // Function to get the size of the queue
    int size() {
        return currentSize;
    }
};
```

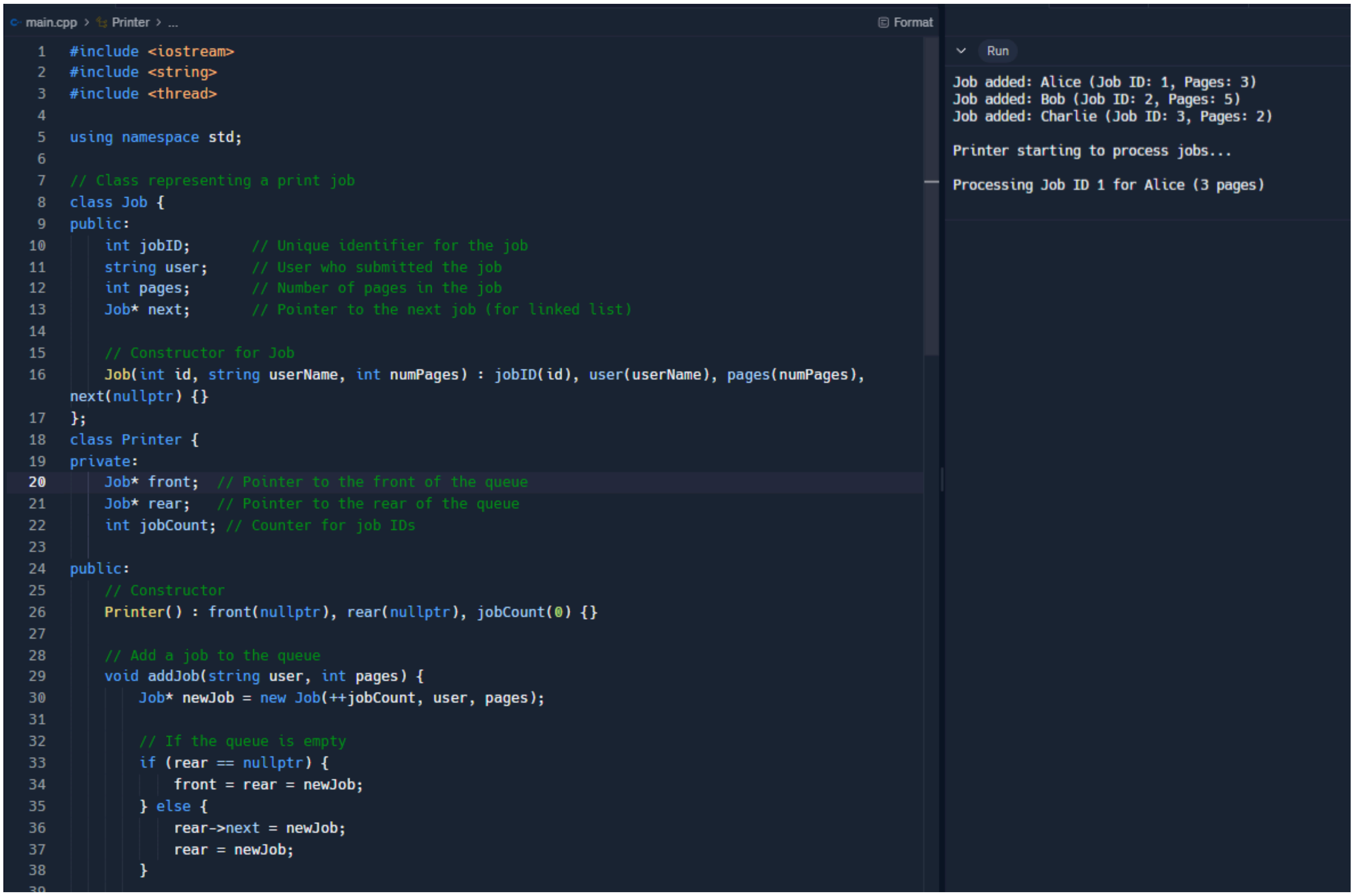
```
Operation 1: Inserting into an empty queue
Inserted: Alice
Queue size: 1

Operation 2: Inserting into a non-empty queue
Inserted: Bob
Queue size: 2

Operation 3: Deleting from a queue with more than one item
Front of the queue before deletion: Alice
Deleted: Alice
Queue size: 1
Front of the queue after deletion: Bob

Operation 4: Deleting from a queue with one item
Front of the queue before deletion: Bob
Deleted: Bob
Queue is now empty: True
```



```cpp
int main() {
    // Create a queue of capacity 5
    Queue studentsQueue(5);

    // Operation 1: Enqueue into an empty queue
    cout << "Operation 1: Enqueue into an empty queue\n";
    studentsQueue.enqueue("Alice");
    cout << "Queue size: " << studentsQueue.size() << "\n\n";

    // Operation 2: Enqueue into a non-empty queue
    cout << "Operation 2: Enqueue into a non-empty queue\n";
    studentsQueue.enqueue("Bob");
    cout << "Queue size: " << studentsQueue.size() << "\n\n";

    // Operation 3: Dequeue from a queue with more than one item
    cout << "Operation 3: Dequeue from a queue with more than one item\n";
    cout << "Front of the queue before dequeue: " << studentsQueue.getFront() << "\n";
    studentsQueue.dequeue();
    cout << "Queue size: " << studentsQueue.size() << "\n";
    cout << "Front of the queue after dequeue: " << studentsQueue.getFront() << "\n\n";

    // Operation 4: Dequeue from a queue with one item
    cout << "Operation 4: Dequeue from a queue with one item\n";
    cout << "Front of the queue before dequeue: " << studentsQueue.getFront() << "\n";
    studentsQueue.dequeue();
    cout << "Queue is now empty: " << (studentsQueue.isEmpty() ? "True" : "False") << "\n";

    return 0;
}
```

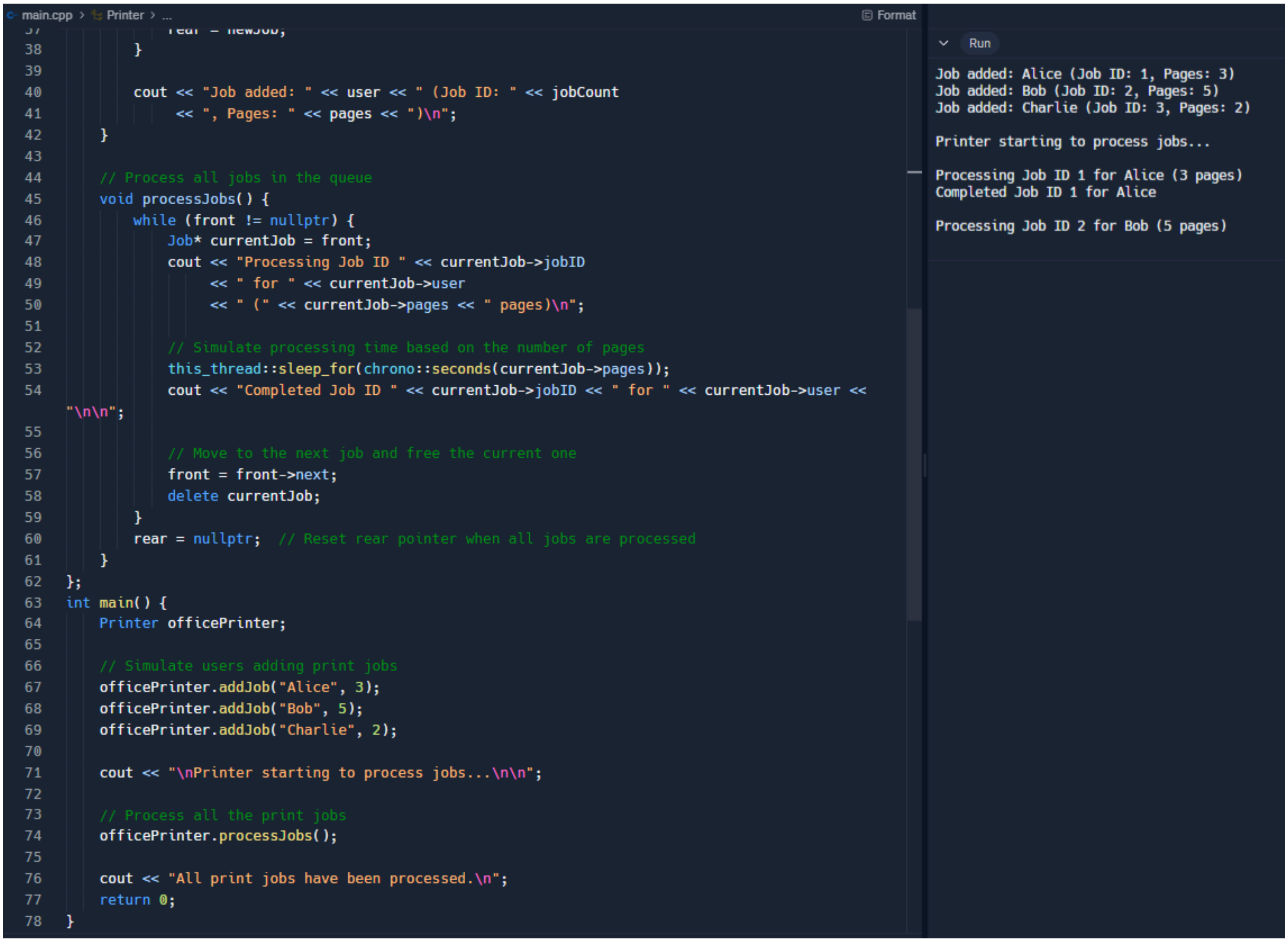
```
Operation 1: Inserting into an empty queue
Inserted: Alice
Queue size: 1

Operation 2: Inserting into a non-empty queue
Inserted: Bob
Queue size: 2

Operation 3: Deleting from a queue with more than one item
Front of the queue before deletion: Alice
Deleted: Alice
Queue size: 1
Front of the queue after deletion: Bob

Operation 4: Deleting from a queue with one item
Front of the queue before deletion: Bob
Deleted: Bob
Queue is now empty: True
```

7. Supplementary Activity



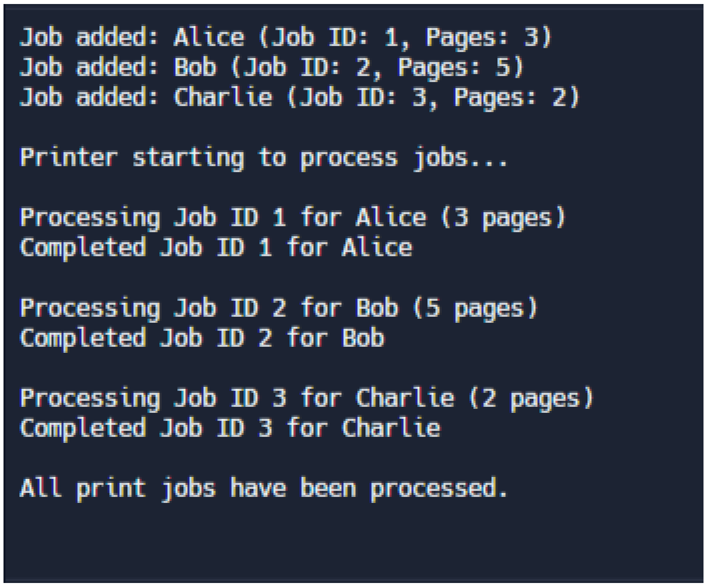
```cpp
#include <iostream>
#include <string>
#include <thread>

using namespace std;

// Class representing a print job
class Job {
public:
    int jobID;        // Unique identifier for the job
    string user;      // User who submitted the job
    int pages;        // Number of pages in the job
    Job* next;        // Pointer to the next job (for linked list)

    // Constructor for Job
    Job(int id, string userName, int numPages) : jobID(id), user(userName), pages(numPages), next(nullptr) {}
};
class Printer {
private:
    Job* front;  // Pointer to the front of the queue
    Job* rear;   // Pointer to the rear of the queue
    int jobCount; // Counter for job IDs

public:
    // Constructor
    Printer() : front(nullptr), rear(nullptr), jobCount(0) {}

    // Add a job to the queue
    void addJob(string user, int pages) {
        Job* newJob = new Job(++jobCount, user, pages);

        // If the queue is empty
        if (rear == nullptr) {
            front = rear = newJob;
        } else {
            rear->next = newJob;
            rear = newJob;
        }
```

```
Job added: Alice (Job ID: 1, Pages: 3)
Job added: Bob (Job ID: 2, Pages: 5)
Job added: Charlie (Job ID: 3, Pages: 2)

Printer starting to process jobs...

Processing Job ID 1 for Alice (3 pages)
```



```cpp
            rear = newJob;
        }

        cout << "Job added: " << user << " (Job ID: " << jobCount
            << ", Pages: " << pages << ")\n";
    }

    // Process all jobs in the queue
    void processJobs() {
        while (front != nullptr) {
            Job* currentJob = front;
            cout << "Processing Job ID " << currentJob->jobID
                 << " for " << currentJob->user
                 << " (" << currentJob->pages << " pages)\n";

            // Simulate processing time based on the number of pages
            this_thread::sleep_for(chrono::seconds(currentJob->pages));
            cout << "Completed Job ID " << currentJob->jobID << " for " << currentJob->user << "\n\n";

            // Move to the next job and free the current one
            front = front->next;
            delete currentJob;
        }
        rear = nullptr;  // Reset rear pointer when all jobs are processed
    }
};
int main() {
    Printer officePrinter;

    // Simulate users adding print jobs
    officePrinter.addJob("Alice", 3);
    officePrinter.addJob("Bob", 5);
    officePrinter.addJob("Charlie", 2);

    cout << "\nPrinter starting to process jobs...\n\n";

    // Process all the print jobs
    officePrinter.processJobs();

    cout << "All print jobs have been processed.\n";
    return 0;
}
```

```
Job added: Alice (Job ID: 1, Pages: 3)
Job added: Bob (Job ID: 2, Pages: 5)
Job added: Charlie (Job ID: 3, Pages: 2)

Printer starting to process jobs...

Processing Job ID 1 for Alice (3 pages)
Completed Job ID 1 for Alice

Processing Job ID 2 for Bob (5 pages)
```



```
Job added: Alice (Job ID: 1, Pages: 3)
Job added: Bob (Job ID: 2, Pages: 5)
Job added: Charlie (Job ID: 3, Pages: 2)

Printer starting to process jobs...

Processing Job ID 1 for Alice (3 pages)
Completed Job ID 1 for Alice

Processing Job ID 2 for Bob (5 pages)
Completed Job ID 2 for Bob

Processing Job ID 3 for Charlie (2 pages)
Completed Job ID 3 for Charlie

All print jobs have been processed.
```

8. Conclusion

I learned how to create a queue system using linked lists in C++. The process involved making a Job class to represent print jobs and a Printer class to manage the job queue. Using linked lists allowed for efficient job management and easy insertion and deletion. The activity simulated real-time printing effectively. I feel I did well overall, but I could improve error handling and user interface design. This exercise helped me understand how queues work in practical situations.

9. Assessment Rubric