Activity No. 5

QUEUES

Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 10/07/2024
Section: CPE21S4	**Date Submitted:** 10/07/2024
Name(s): SANCHEZ, Christan Ray R.	**Instructor:** Prof. Ma. Rizette Sayo

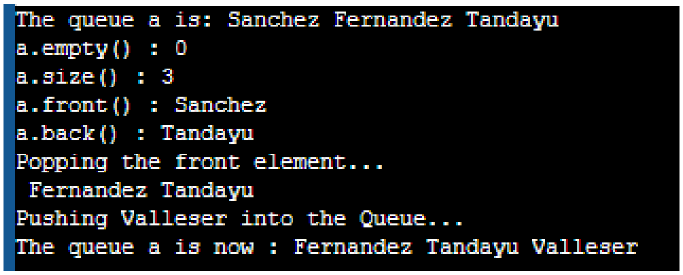
6. Output

Table 5-1

SOURCE CODE

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

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

int main() {
    std::queue<std::string> a;

    // Adding elements to the queue
    a.push("Sanchez");
    a.push("Fernandez");
    a.push("Tandayu");

    // Displaying the queue contents
    std::cout << "The queue a is:";
    display(a);

    // Displaying the properties of the queue
    std::cout << "a.empty() : " << a.empty() << "\n";
    std::cout << "a.size() : " << a.size() << "\n";
    std::cout << "a.front() : " << a.front() << "\n";
    std::cout << "a.back() : " << a.back() << "\n";

    // Popping the front element and displaying the modified queue
    std::cout << "Popping the front element...\n";
    a.pop();
    display(a);

    std::cout<<"Pushing Valleser into the Queue...\n";
    // Pushing a new element and displaying the modified queue
    a.push("Valleser");
    std::cout << "The queue a is now :";
    display(a);

    return 0;
}
```

OUTPUT



```
The queue a is: Sanchez Fernandez Tandayu
a.empty() : 0
a.size() : 3
a.front() : Sanchez
a.back() : Tandayu
Popping the front element...
 Fernandez Tandayu
Pushing Valleser into the Queue...
The queue a is now : Fernandez Tandayu Valleser
```

OBSERVATIONS

I noticed that the sample code was specifically for integers, so I modified it to use strings for entering the queue. I also followed the instructions for passing an array of students.

Table 5-2

SOURCE CODE

```
#include <iostream>

struct Node {
        std::string data;
        Node* next;

        Node(const std::string& val) : data(val), next(nullptr) {}
};

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

public:
        Queue() : front(nullptr), rear(nullptr) {}

        // Inserting an item into a non-empty queue
        void enqueue(const std::string& value) {
                Node* newNode = new Node(value);
                if (rear) {
                        rear->next = newNode; // Link new node at the end of the queue
        }
        rear = newNode; // Update rear
        if (!front) {
            front = newNode; // If the queue was empty, front also points to the new node
        }
    }

    // Inserting an item into an empty queue
    void enqueueEmpty(const std::string& value) {
        Node* newNode = new Node(value);
        front = rear = newNode; // Both front and rear point to the new node
    }

    // Deleting an item from a queue with more than one item
    void dequeueMultiple() {
        if (!front) return; // If the queue is empty, do nothing
        Node* temp = front;
        front = front->next; // Move front to the next node
        delete temp; // Delete the old front
        if (!front) {
            rear = nullptr; // If the queue is empty after the operation
        }
    }

    // Deleting an item from a queue with one item
    void dequeueSingle() {
        if (!front) return; // If the queue is empty, do nothing
        delete front;
        front = rear = nullptr;
    }

    // Display the queue
    void display() {
        Node* current = front;
        while (current) {
            std::cout << current->data << " ";
            current = current->next;
        }
        std::cout << "\n";
    }
```

```
        ~Queue() {
                while (front) {
                        dequeueMultiple();
                }
        }
};

int main() {
        Queue queue;

        // Inserting items
        std::cout << "Inserting 'Christan' into the empty queue:\n";
        queue.enqueueEmpty("Christan");
        queue.display();

        std::cout << "Inserting 'Kuristan' into the non-empty queue:\n";
        queue.enqueue("Kuristan");
        queue.display();

        std::cout << "Inserting 'Chocnut' into the non-empty queue:\n";
        queue.enqueue("Chocnut");
        queue.display();

        // Deleting items
        std::cout << "Deleting an item from the queue (more than one item):\n";
        queue.dequeueMultiple();
        queue.display();

        std::cout << "Deleting the last item from the queue:\n";
        queue.dequeueMultiple();
        queue.display();

        std::cout << "Deleting the last remaining item:\n";
        queue.dequeueSingle();
        queue.display();

        return 0;
}
```

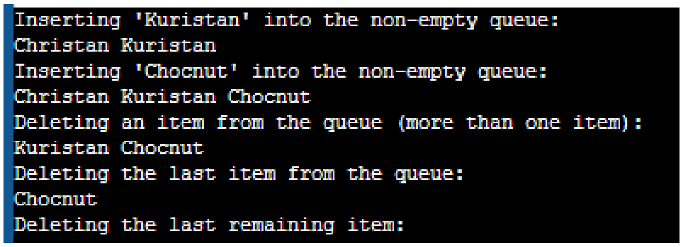
OUTPUT	 Inserting 'Kuristan' into the non-empty queue: Christan Kuristan Inserting 'Chocnut' into the non-empty queue: Christan Kuristan Chocnut Deleting an item from the queue (more than one item): Kuristan Chocnut Deleting the last item from the queue: Chocnut Deleting the last remaining item:
Inserting 'Christan’ into the empty queue	Christan
Inserting 'Kuristan' into the non-empty queue	Christan Kuristan
Inserting 'Chocnut' into the non-empty queue	Christan Kuristan Chocnut
Deleting an item from the queue (more than one item)	Kurdistan Chocnut
Deleting the last item from the queue	Chocnut
Deleting the last remaining item	(EMPTY)

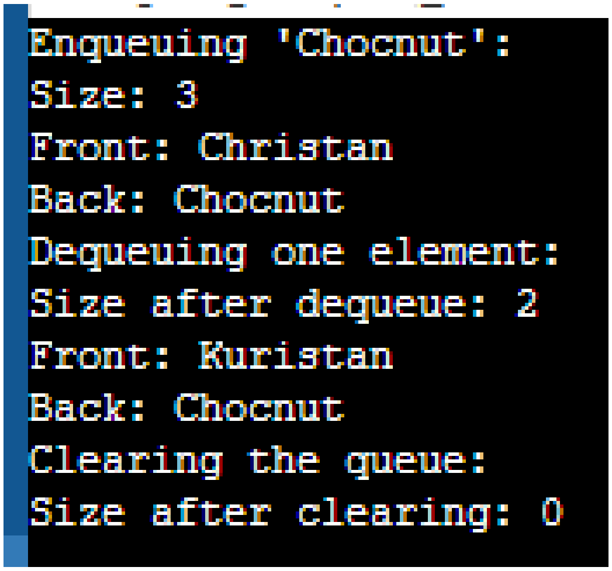
Table 5-3

SOURCE CODE

```
#include <iostream>
#include <stdexcept>

class CircularQueue {
private:
        std::string* q_array;  // Pointer to the queue array
        int q_capacity;        // Capacity of the queue
        int q_size;            // Current size of the queue
        int q_front;           // Index of the front item
        int q_back;            // Index of the back item

public:
        // Constructor
        CircularQueue(int capacity) : q_capacity(capacity), q_size(0), q_front(0), q_back(-1) {
                q_array = new std::string[q_capacity]; // Dynamically allocate the queue array
        }

        // Destructor
        ~CircularQueue() {
                delete[] q_array; // Deallocate the queue array
        }
```

```
// Check if the queue is empty
bool empty() const {
        return q_size == 0;
}

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

// Clear the queue
void clear() {
        q_size = 0;
        q_front = 0;
        q_back = -1;
}

// Access the front element
std::string front() const {
        if (empty()) throw std::runtime_error("Queue is empty.");
        return q_array[q_front];
}

// Access the back element
std::string back() const {
        if (empty()) throw std::runtime_error("Queue is empty.");
        return q_array[q_back];
}

// Enqueue an element
void enqueue(const std::string& value) {
        if (q_size == q_capacity) throw std::runtime_error("Queue is full.");
        q_back = (q_back + 1) % q_capacity; // Move back circularly
        q_array[q_back] = value; // Insert the new value
        q_size++; // Increase the size
}

// Dequeue an element
void dequeue() {
        if (empty()) throw std::runtime_error("Queue is empty.");
        q_front = (q_front + 1) % q_capacity; // Move front circularly
        q_size–; // Decrease the size
}

// Copy Constructor
CircularQueue(const CircularQueue& other) {
        q_capacity = other.q_capacity;
        q_size = other.q_size;
        q_front = other.q_front;
        q_back = other.q_back;
        q_array = new std::string[q_capacity];
        for (int i = 0; i < q_size; ++i) {
            q_array[(q_front + i) % q_capacity] =
other.q_array[(other.q_front + i) % other.q_capacity];
        }
    }

    // Copy Assignment Operator
    CircularQueue& operator=(const CircularQueue& other) {
        if (this != &other) {
            delete[] q_array; // Deallocate existing array
            q_capacity = other.q_capacity;
            q_size = other.q_size;
            q_front = other.q_front;
            q_back = other.q_back;
            q_array = new std::string[q_capacity];
            for (int i = 0; i < q_size; ++i) {
                q_array[(q_front + i) % q_capacity] =
other.q_array[(other.q_front + i) % other.q_capacity];
            }
        }
        return *this;
    }
};

int main() {
    CircularQueue queue(5); // Create a queue with capacity 5

    // Enqueue operations
    std::cout << "Enqueuing 'Christan':\n";
    queue.enqueue("Christan");
    std::cout << "Size: " << queue.size() << "\n";

    std::cout << "Enqueuing 'Kuristan':\n";
    queue.enqueue("Kuristan");
    std::cout << "Size: " << queue.size() << "\n";

    std::cout << "Enqueuing 'Chocnut':\n";
    queue.enqueue("Chocnut");
    std::cout << "Size: " << queue.size() << "\n";

    // Front and Back access
    std::cout << "Front: " << queue.front() << "\n";
    std::cout << "Back: " << queue.back() << "\n";

    // Dequeue operations
    std::cout << "Dequeuing one element:\n";
    queue.dequeue();
    std::cout << "Size after dequeue: " << queue.size() << "\n";

    std::cout << "Front: " << queue.front() << "\n";
    std::cout << "Back: " << queue.back() << "\n";

    // Clear the queue
    std::cout << "Clearing the queue:\n";
    queue.clear();
    std::cout << "Size after clearing: " << queue.size() << "\n";

    return 0;
}
```

OUTPUT



```
Enqueuing 'Chocnut':
Size: 3
Front: Christan
Back: Chocnut
Dequeuing one element:
Size after dequeue: 2
Front: Kuristan
Back: Chocnut
Clearing the queue:
Size after clearing: 0
```

7. Supplementary Activity

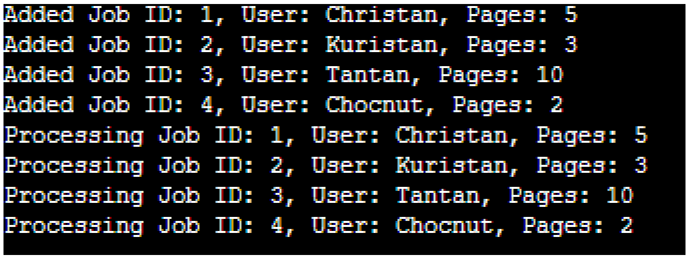
SOURCE CODE

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

class Job {
public:
    int jobId;
    std::string userName;
    int numPages;
    Job* next;

    // Constructor
    Job(int id, const std::string& user, int pages)
```

```
        : jobId(id), userName(user), numPages(pages), next(nullptr) {}
};


class Printer {
private:
    Job* front; // Pointer to the front of the queue
    Job* rear;  // Pointer to the rear of the queue

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

    // Add a job to the queue
    void addJob(int id, const std::string& user, int pages) {
        Job* newJob = new Job(id, user, pages);
        if (rear == nullptr) { // If the queue is empty
            front = rear = newJob;
        } else {
            rear->next = newJob; // Link the new job
            rear = newJob;       // Move the rear pointer
        }
        std::cout << "Added Job ID: " << id << ", User: " << user << ",
Pages: " << pages << "\n";
    }

    // Process all jobs in the queue
    void processJobs() {
        while (front != nullptr) {
            Job* temp = front; // Get the job at the front
            front = front->next; // Move the front pointer
            std::cout << "Processing Job ID: " << temp->jobId
               << ", User: " << temp->userName
               << ", Pages: " << temp->numPages << "\n";
            delete temp; // Free the memory
        }
        rear = nullptr; // Reset rear pointer
    }

    // Destructor to clean up any remaining jobs
    ~Printer() {
        while (front != nullptr) {
            Job* temp = front;
            front = front->next;
            delete temp;
        }
    }
};
```

```
int main() {
        Printer printer;

        // Simulate adding jobs
        printer.addJob(1, "Christan", 5);
        printer.addJob(2, "Kuristan", 3);
        printer.addJob(3, "Tantan", 10);
        printer.addJob(4, "Chocnut", 2);

        // Process the jobs
        printer.processJobs();

        return 0;
}
```

OUTPUT



```
Added Job ID: 1, User: Christan, Pages: 5
Added Job ID: 2, User: Kuristan, Pages: 3
Added Job ID: 3, User: Tantan, Pages: 10
Added Job ID: 4, User: Chocnut, Pages: 2
Processing Job ID: 1, User: Christan, Pages: 5
Processing Job ID: 2, User: Kuristan, Pages: 3
Processing Job ID: 3, User: Tantan, Pages: 10
Processing Job ID: 4, User: Chocnut, Pages: 2
```

8. Conclusion

In this activity, I gained valuable insights into manipulating queues, which will be beneficial for future programming tasks. The simulation of a shared printer using a queue allowed me to understand the fundamental concepts of queue operations, such as adding and processing jobs in a first-come, first-served manner. Additionally, I learned the importance of careful input selection. For instance, when adapting the code for handling students' names, I realized the necessity of transitioning from integers to strings to accommodate the different data types. Overall, this experience has enhanced my understanding of data structures and their applications, equipping me with skills that will prove useful in future projects.

9. Assessment Rubric