CC-213L

Data Structures and Algorithms

Laboratory 05

Queue ADT

Version: 1.0.0

Release Date: 18-10-2024

Department of Information Technology

University of the Punjab

Lahore, Pakistan

Contents:

Learning Objectives:

- Queue ADT
- PriorityQueue
- Double Ended Queue
- Applications of Queue ADT

Resources Required:

- Desktop Computer or Laptop
- Microsoft ® Visual Studio 2022

General Instructions:

- In this Lab, you are **NOT** allowed to discuss your solution with your colleagues, even not allowed to ask how is s/he doing, this may result in negative marking. You can **ONLY** discuss with your Teaching Assistants (TAs) or Lab Instructor.
- Your TAs will be available in the Lab for your help. Alternatively, you can send your queries via email to one of the following.

Teachers:		
Course/Lab Instructor	Prof. Dr. Syed Waqar ul Qounain	swjaffry@pucit.edu.pk
Teacher Assistants	Tahir Mustafvi	bcsf20m018@pucit.edu.pk
	Maryam Rasool	bcsf21m055@pucit.edu.pk

Background and Overview:

Queue:

A queue is a linear data structure that operates on the principle of First-In-First-Out (FIFO). Think of it as similar to a real-world queue, like people standing in line. In a queue, elements are added at the back, a process known as enqueueing, and removed from the front, referred to as dequeuing. This ensures that the element added first will be the first one to be removed. Let's explore this with an analogy:

Adding an Element to the Queue: Imagine a queue as a line of people waiting for something, like ordering food at a food truck. There are already a few people in the line, and you're joining at the back. When you join, you become the last person in line. Similarly, when you add an element to a queue, it goes to the end of the queue.

For example, if the queue contains numbers like [3, 7, 10], and you want to add the number 15, it will be placed at the end of the queue. So, the updated queue would look like this: Queue: [3, 7, 10, 15]

Removing an Element from the Queue: Continuing with the food truck analogy, let's say it's your turn to order. You step forward, place your order, and then move away from the line. Now, the person who was directly behind you moves up to the front to place their order. In a queue, when you remove an element, the element at the front is taken out, and the element that was behind it becomes the new front.

For instance, if the queue is [3, 7, 10, 15], and you remove the first element, which is 3, the updated queue will look like this: Queue: [7, 10, 15] The element 7, originally second in line, is now at the front.

Implementation Details: To understand how a queue operates, envision a row of boxes and two individuals, Mr. Rear and Mr. Front. Each box represents a spot in the queue where items can be placed.

Mr. Rear stands where the last filled box is located. He knows precisely where the last item was positioned. When you want to add a new item to the line of boxes, you simply ask Mr. Rear. He will instruct you on which box to place the new item in. Once you've added the item, Mr. Rear steps forward, prepared to guide the placement of the next item.

Mr. Front stands where the first filled box is. His role is to inform you which item should be taken out next. When you need to use or remove an item from the line, you ask Mr. Front. He points to the item in the first box and indicates that this is the one you should remove. After you've taken it out, Mr. Front moves to the next box, ready for the next time you need to remove an item.

Working Together: The collaboration between Mr. Rear and Mr. Front ensures that the queue remains in order. As you add new items, Mr. Rear assists you in placing them correctly at the end of the line, effectively extending the queue. When you need to use an item, Mr. Front indicates which item is up next, maintaining the First-In-First-Out (FIFO) principle of the queue.

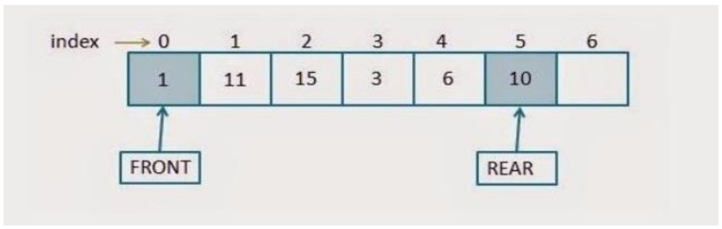

Figure 1(Queue)

When Rear reaches the end of the queue: Let's consider a scenario with 10 boxes. Initially, we placed items in all 10 boxes. Now, we've removed three items, specifically from boxes 1, 2, and 3. Mr. Front is now positioned at box number 4. The filled boxes are 4, 5, 6, 7, 8, 9, and 10, with 1, 2, and 3 being empty. Now, the question arises: What should we do when we want to add a new item? Should we increase the number of boxes to accommodate new items? But what about the empty boxes 1, 2, and 3?

Instead of expanding the number of boxes, here's a more efficient approach: Place the new item in box number 1 and move Mr. Rear from box number 10 to box number 1. This allows us to utilize the previously empty boxes 1, 2, and 3 for the new items.

Key Takeaways:

- When an item is removed, the front is advanced, and the number of elements is decreased.
- When a new item is inserted, the rear is advanced, and the number of elements is incremented.
- The array is resized when the number of elements matches the capacity.
- The data is stored between the front and rear pointers, creating a continuous array with the front and rear ends connected.

To visualize the underlying data structure of a queue, imagine an array where one end is connected to the other, forming a circular structure.

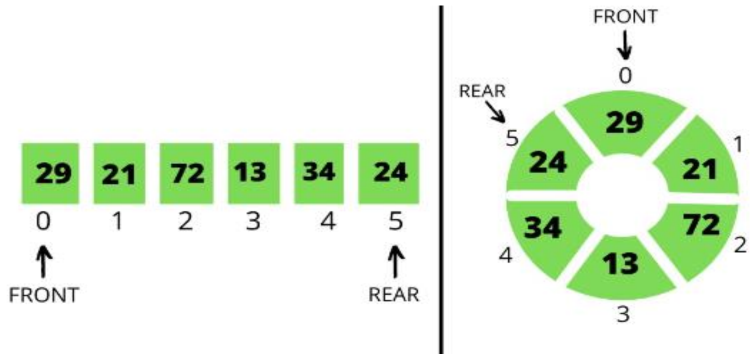

Figure 2(Circular Queue)

Queue Applications:

Queue data structures are widely used in various applications and scenarios where managing data and processes in a first-in, first-out (FIFO) manner is essential. Here are different types of applications where the queue data structure is commonly employed.

Operating Systems and Task Management:

1. Task Scheduling: Operating systems use queues to schedule and prioritize tasks, ensuring that tasks are executed based on their priority and order of arrival.
2. Thread Management: Multithreaded applications use queues to manage and coordinate threads for efficient task execution.
3. Real-Time Systems: Queues play a crucial role in real-time systems, ensuring that events are processed in the order they occur.

Document and Data Management:

1. Print Queues: Print jobs are managed using queues, allowing documents to be printed in the order they are received.
2. Print Spooling: Print spoolers use queues to manage and prioritize print jobs.
3. Buffer Management: Queues are used to manage data buffers, preventing overflow and ensuring controlled data transmission.
4. Data Streaming: In data streaming applications, queues are used to buffer and process data, ensuring a smooth flow of information.

Customer Service and Communication:

1. Call Center Systems: Call centers use queues to manage incoming customer calls and direct them to available agents.
2. Request Handling: Web servers and application servers use queues to manage incoming requests, such as HTTP requests.
3. Message Queues: Message queuing systems enable communication between distributed applications and services in a decoupled manner.

E-commerce and Inventory Management:

1. Order Processing: E-commerce websites use queues to manage incoming orders, ensuring they are processed in sequence.
2. Inventory Management: Retail and supply chain systems use queues to manage inventory, ensuring efficient restocking and shipping.

Data Structures and Algorithms:

1. Data Structures: Queues serve as fundamental data structures and are often used as building blocks for more complex data structures like double-ended queues (deques).
2. Breadth-First Search (BFS): BFS traversal in graph algorithms utilizes a queue to explore nodes level by level.
3. Simulation and Modeling: Queues are used in simulations to model processes, such as traffic flow, customer service, and manufacturing.

Transportation and Traffic Control:

1. Traffic Management: Traffic control systems use queues to manage the flow of vehicles and prevent congestion.

These categories demonstrate the widespread utility of the queue data structure in various domains, emphasizing its role in maintaining the order and priority of tasks and data elements.

Activities

Pre-Lab Activities:

Task 01 Implementation of Linear Queue

Queues are fundamental data structures used in various algorithms and applications. They can be created using arrays or linked lists. However, for our current focus, we will concentrate on implementing a queue using arrays, as we haven't covered linked lists yet.

To gain a deeper understanding of how the queue data structure operates, you will be tasked with implementing the **MyQueue** Abstract Data Type (ADT) in C++. This assignment is part of your grading and will be assessed online in the classroom. It's essential to ensure that your implementation is accurate, handles all possible scenarios and corner cases, as this queue will be utilized in solving problems during in-lab tasks. Any issues in your queue ADT may pose challenges when working on problems that involve queue data structures, as the use of the STL library is not permitted. You are supposed to add time and space complexity of each function in comments.

```
template<typename T>
class myLinearQueue {
   int rearIndex;                    // Index of the rear element
   int frontIndex;                   // Index of the front element
   int queueCapacity;                // Maximum capacity of the queue
   int numberOfElements;             // Number of elements in the queue
   T * queueData;                    // Array to store queue elements
   void resize(int newSize);         // Private helper method for resizing the queue

public:
   myLinearQueue() {
      rearIndex = frontIndex = numberOfElements = queueCapacity = 0;
      queueData = nullptr;
   }

   myLinearQueue(const MyQueue<T> &);                        // Copy constructor
   myLinearQueue<T> & operator=(const MyQueue<T> &);         // Assignment operator
   ~MyQueue();                                                // Destructor

   void enqueue(const T element);    // Add an element to the back of the queue
   T dequeue();                      // Remove and return the front element
   T getFront() const;               // Get the front element without removing it
   bool isEmpty() const;             // Check if the queue is empty
   bool isFull() const;              // Check if the queue is full
   int size() const;                 // Get the current number of elements
   int getCapacity() const;          // Get the maximum capacity of the queue
   };
```

Task 02: Implementation of Circular Queue

A Circular Queue ADT is a data structure that works as a regular queue but with a circular wrap-around concept. It efficiently uses space by treating the queue as circular, meaning after the last element, it loops back to the front if space is available. You are supposed to add time and space complexity of each function in comments.
In this task, we will implement it using **templates** in C++ to allow flexibility with the data types.

```
template <class T>
class CircularQueue {
/***** define data members required to implement CircularQueue *****/
private:
    int front;          // Points to the front element of the queue
    int rear;           // Points to the last element in the queue
    int capacity;       // Maximum size of the queue
    int count;          // Current size of the queue
    T* queueArray;      // Array to hold queue elements

public:
    CircularQueue(int size);   // Constructor to initialize the queue with a given size
    ~CircularQueue();          // Destructor to free memory
    void enqueue(const T& data);  // Function to add an element to the queue
    T dequeue();               // Function to remove and return the front element
    bool isEmpty() const;      // Check if the queue is empty
    bool isFull() const;       // Check if the queue is full
    int size() const;          // Return the current size of the queue
    T peek() const;            // Peek at the front element without removing it
};
```

Explanation:

- The CircularQueue class uses C++ templates to handle elements of different data types.
- The constructor initializes a circular queue with a specified size.
- The destructor releases any dynamically allocated memory when the queue is destroyed.
- The enqueue function adds an element to the queue at the rear, and the position is updated circularly.
- The dequeue function removes and returns the front element, and the front pointer is updated circularly.
- The isEmpty function checks if the queue is empty.
- The isFull function checks if the queue is full.
- The size function returns the current number of elements in the queue.
- The peek function allows you to view the front element without dequeuing it.

Example Usage:

```
CircularQueue<int> intQueue(3);
intQueue.enqueue(10);
intQueue.enqueue(20);
intQueue.enqueue(30);

intQueue.dequeue();  // Removes 10
intQueue.enqueue(40);  // 40 gets added after 30 (circular)


CircularQueue<std::string> stringQueue(2);
stringQueue.enqueue("hello");
stringQueue.enqueue("world");
stringQueue.dequeue();  // Removes "hello"
stringQueue.enqueue("again");
```

In-Lab Activities

Task 01 Implementation of Priority Queue

A Priority Queue ADT is a data structure that combines the features of a queue with the added concept of priorities. In a Priority Queue, elements are enqueued based on their priority, and when dequeued, the element with the highest priority is removed first. This ADT is implemented using templates in C++ to allow flexibility with the data types.

```
template <class T>
class PriorityQueue {
        /***** define data members required to implement PriorityQueue *****/
public:
   PriorityQueue();                          // Constructor to initialize the priority queue
   ~PriorityQueue();                         // Destructor to free memory if necessary
   void enqueue(const T& data, int priority); // Function to enqueue an element with a given priority
   T dequeue();                              // Function to dequeue the element with the highest priority
   bool isEmpty() const;                     // Function to check if the priority queue is empty
   bool isFull() const;                      // Function to check if the priority queue is full
   int size() const;                         // Function to get the size of the priority queue
};
```

Explanation:

- The **PriorityQueue** class is designed using C++ templates to make it generic and capable of handling elements of different data types.
- The constructor initializes an empty priority queue.
- The destructor can be implemented to release any allocated memory when needed.
- The **enqueue** function adds an element with a specified data value and priority. Lower values of priority indicate higher priority.
- The **dequeue** function removes and returns the element with the highest priority from the queue.
- The **isEmpty** function checks whether the priority queue is empty.
- The **isFull** function checks whether the priority queue is full.
- The **size** function returns the current size of the priority queue.

Usage:
Here's how you can use the **PriorityQueue** ADT in C++ with different data types:

```
PriorityQueue<int> intQueue;
intQueue.enqueue(42, 2);
intQueue.enqueue(36, 1);
intQueue.enqueue(17, 3);

PriorityQueue<std::string> stringQueue;
stringQueue.enqueue("apple", 5);
stringQueue.enqueue("banana", 3);
stringQueue.enqueue("cherry", 7);

int highestPriorityInt = intQueue.dequeue();
```

```
std::string highestPriorityStr = stringQueue.dequeue();

// Now, highestPriorityInt contains 17 (highest priority)
// highestPriorityStr contains "cherry" (highest priority)
```

Task 02 Double Ended Queue

Double-ended queue is almost like the queue that we discussed earlier with one key difference that it allows insertion and removal of elements from both ends. This means you can add elements not only to the back (rear) of the queue but also to the front. Similarly, you can remove elements not only from the front but also from the back. Imagine it as a line where people can join from back or front and can also leave from back or front.

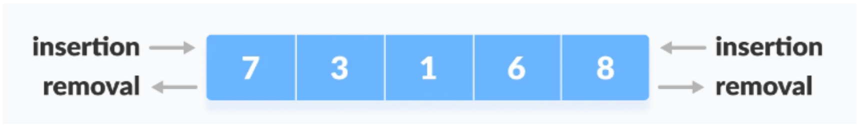

Figure 3(Double Ended Queue)

To insert an element at the front end:

1. Resize the deque if it's full.
2. If the front is equal to 0 or at the initial position:
   2.1. Move the front to point to the last index of the array.
3. Else:
   3.1. Decrement the front by 1.
   3.2. Push the current Value into Arr [front] = Value.
4. The rear remains the same.

To insert an element at the rear end:

1. Check if the deque is full.
2. If the rear is equal to the last index of the array (size - 1):
   2.1. Reinitialize rear to 0.
3. Else:
   3.1. Increment rear by 1.
   3.2. Push the current Value into Arr [rear] = Value.
4. Front remains the same.

To delete element from the rear end:

1. Check if the deque is empty or not.
2. If the deque has only one element:
   2.1. Set front to -1.
   2.2. Set rear to -1.
3. Else, if the rear points to the first index of the array:
   3.1. Move rear to point to the last index (size - 1) of the array.
4. Otherwise:
   4.1. Decrement rear by 1 (rear = rear - 1).

To delete an element from the front end:

1. Check if the deque is empty or not.
2. If the deque has only one element:
   2.1. Set front to -1.
   2.2. Set rear to -1.

3. Else, if front points to the last index of the array:
    3.1. Move front to point to the first index of the array (front = 0).
4. Otherwise:
    4.1. Increment front by 1 (front = front + 1).

Let's demonstrate these operations with a dry run example: Suppose we have an empty deque initially. We perform the following operations:

1. InsertFront(5)
    - Deque: (5)
2. InsertRear(10)
    - Deque: (5, 10)
3. InsertFront(2)
    - Deque: (2, 5, 10)
4. RemoveFront()
    - Deque: (5, 10)
5. RemoveRear()
    - Deque: (5)
6. InsertRear(7)
    - Deque: (5, 7)

This example illustrates the behavior of a deque as we insert and remove elements from both the front and rear ends.

Task 03: FIFO Page Replacement Algorithm

Background

Paging is an essential memory management scheme in modern operating systems that allows efficient use of memory by breaking down large blocks of data into smaller, manageable "pages." When a program is executed, its data is divided into fixed-sized pages, which are loaded into the computer’s memory (RAM). The operating system is responsible for managing these pages and deciding which ones to load into memory when needed.
However, when the memory is full and a new page needs to be loaded, one of the pages currently in memory must be replaced. This scenario is known as a page fault. The operating system uses page replacement algorithms to decide which page to remove. One of the simplest page replacement algorithms is the FIFO (First In, First Out) algorithm, which removes the oldest page first—just like a queue.
In this task, you will implement the FIFO page replacement algorithm to simulate how the operating system handles page requests, page faults, and page hits.

Explanation of FIFO Page Replacement Algorithm

- **Queue Mechanism**: The FIFO page replacement algorithm operates using a queue to store the pages currently in memory. The first page to be added is the first one to be removed when a new page needs to be loaded, and the memory is full.

- **Page Fault:** A page fault occurs when the requested page is not currently loaded in memory. This results in the operating system loading the new page and possibly replacing an old one.

- **Page Hit:** If the requested page is already in memory, it is considered a hit, and no replacement is necessary.

Example

Consider the following sequence of page requests:
1, 3, 0, 3, 5, 6
Assume the available memory can hold 3 pages at a time.

1. Pages 1, 3, and 0 are loaded, causing 3 page faults.
2. When page 3 is requested again, it is already in memory, resulting in a page hit.
3. Page 5 is requested, causing a page fault. Page 1 (the oldest) has been replaced.
4. Page 6 is requested, causing another page fault. Page 3 (now the oldest) has been replaced.

In total, this sequence generates 5 page faults and 1 page hit.

Input/Output Format

Input:

- Memory queue size: an integer representing the number of pages the memory can hold.
- A sequence of page requests entered by the user, terminated by "quit."

Output:

- After each request, display whether it was a page hit or a page fault.
- After each request, display the current state of the memory queue.
- Finally, display the total number of page hits and page faults.

Task 04: Possible Binary Numbers

Given a number N, your task is to use a queue data structure to print all possible binary numbers with decimal values from 1 to N. For instance, if the input is 4, the expected output would be: 1, 10, 11, 100. Here's an intriguing approach that employs a queue data structure to achieve this:

1. Begin with an empty queue of strings.
2. Add the first binary number, "1," to the queue.
3. Remove and print the front element of the queue.
4. Extend the front item by appending "0" to it and place it back in the queue.
5. Extend the front item by appending "1" to it and place it back in the queue.
6. Repeat steps 3 to 5 until you reach the desired end value.

Example Simulation of Queue:

Let's perform a dry run of the given task for N = 4 to see how it generates binary numbers using a queue:

1. Initialize an empty queue: **Queue = []**
2. Add the first binary number, "1," to the queue: **Queue = ["1"]**
3. Remove and print the front element of the queue:
   - Printed: "1"
   - Queue: **Queue = []**
4. Extend the front item by appending "0" to it and place it back in the queue: **Queue = ["10"]**
5. Extend the front item by appending "1" to it and place it back in the queue: **Queue = ["10", "11"]**
6. Repeat steps 3 to 5 until you reach the desired end value.

Iteration 1:

- Remove and print the front element of the queue:
  - Printed: "10"
  - Queue: **Queue = ["11"]**

- Extend the front item by appending "0" to it and place it back in the queue: **Queue = ["11", "100"]**
- Extend the front item by appending "1" to it and place it back in the queue: **Queue = ["11", "100", "101"]**

Iteration 2:

- Remove and print the front element of the queue:
    - Printed: "11"
    - Queue: **Queue = ["100", "101"]**
- Extend the front item by appending "0" to it and place it back in the queue: **Queue = ["100", "101", "110"]**
- Extend the front item by appending "1" to it and place it back in the queue: **Queue = ["100", "101", "110", "111"]**

Iteration 3:

- Remove and print the front element of the queue:
    - Printed: "100"
    - Queue: **Queue = ["101", "110", "111"]**
- Terminate the loop as 100 which equivalents to 4 is printed.

An example of the queue is given as follows

Front	**Queue**				
1					
10	11				
11	100	101			
100	101	110	111		
101	110	111	1000	1001	

Print
1
10
11
100
101

Post-Lab Activities

Task 01: Least Recently Used (LRU) Cache Replacement Algorithm

Background

Computers use cache memory to temporarily store frequently accessed data for faster retrieval. Cache memory is much faster than primary memory but has limited capacity. Therefore, efficient management of the cache is critical, especially when new data needs to be stored, and the cache is full.

A common approach to managing cache memory is the Least Recently Used (LRU) Cache Replacement Algorithm. The LRU algorithm tracks the order in which data is accessed and removes the data that has not been used for the longest time. This approach ensures that the most recently accessed data is readily available, making the system more efficient.

Explanation of LRU Cache

- **Queue Structure:** The LRU Cache behaves like a fixed-size queue where the most recently accessed page is moved to the front. When new data needs to be loaded and the cache is full, the least recently used page (at the rear) is removed.

- **Page Hit:** If a requested page is already present in the cache, it is moved to the front, indicating that it has been accessed recently.

- **Page Fault:** If a requested page is not in the cache and the cache is full, the least recently used page is removed (from the rear), and the new page is inserted at the front.

Example

Consider an LRU Cache with a size of 3, and the following series of page requests:

5, 2, 7, 5, 9, 5, 3

Here's how the LRU cache would operate:

1. Page Request: 5
   The cache is empty, so page 5 is added at the front.
   Cache State: [5]

2. Page Request: 2
   Page 2 is added at the front.
   Cache State: [2, 5]

3. Page Request: 7
   Page 7 is added at the front.
   Cache State: [7, 2, 5]

4. Page Request: 5
   Page 5 is already in the cache, so it's moved to the front.
   Cache State: [5, 7, 2]

5. Page Request: 9
   The cache is full, so the least recently used page (2) is removed, and page 9 is added to the front.
   Cache State: [9, 5, 7]

6. Page Request: 5
   Page 5 is already in the cache, so it's moved to the front.
   Cache State: [5, 9, 7]

7. Page Request: 3
   The cache is full again, so page 7 (the least recently used) is removed, and page 3 is added to the front.
   Cache State: [3, 5, 9]

This process ensures that frequently accessed pages stay in the cache, while the least used ones are evicted when necessary.

Input/Output Format

Input:

- Cache size: an integer representing the number of pages the cache can hold.
- A sequence of page requests entered by the user, terminated by "quit."

Output:

- After each request, display whether it was a page hit or a page fault.
- After each request, display the current state of the cache.
- Finally, display the final state of the cache.

Task 02: Stack Using Queues

Introduction

In computer science, a Stack is a data structure that follows the Last In First Out (LIFO) principle, meaning that the last element added to the stack is the first one to be removed. In this task, we will implement a Stack using two Queues to demonstrate how different data structures can be utilized to achieve the desired functionality.

Objective

The objective of this task is to create a Stack using two Queue data structures. We will design push and pop operations to ensure that Stack maintains its LIFO behavior while leveraging the First In First Out (FIFO) nature of Queues.

Background

- **Stack:** A collection of elements that supports push and pop operations. The most recently added element is accessed first.
- **Queue:** A collection of elements that supports enqueue and dequeue operations. The element added first is accessed first.

Implementation Strategy

To implement the Stack using Queues, we will follow these steps:

1. Two Queues: We will use two queues (queue1 and queue2).
2. Push Operation:
   - Enqueue the new element into queue1.
   - Move all elements from queue1 to queue2, which will reverse the order of the elements.
   - Swap the names of queue1 and queue2 to maintain the state.
3. Pop Operation: Simply dequeue from queue1, which will give us the most recently added element

Submissions:

- For In-Lab Activity:
  - Save the files on your PC.
  - TA’s will evaluate the tasks offline.
- For Pre-Lab & Post-Lab Activity:
  - Submit the .cpp file on Google Classroom and name it to your roll no.

Evaluations Metric:

- All the lab tasks will be evaluated offline by TA’s

• **Division of Pre-Lab marks:**	**[30 marks]**
▪ Task 01: Implementation of Linear Queue	[10 marks]
▪ Task 02: Implementation of Circular Queue	[20 marks]
• **Division of In-Lab marks:**	**[80 marks]**
▪ Task 01: Priority Queue Implementation	[20 marks]
▪ Task 02: Double Ended Queue	[20 marks]
▪ Task 03: FIFO Page Replacement Algorithm	[20 marks]
▪ Task 04: Possible Binary Numbers	[20 marks]
• **Division of Post-Lab marks:**	**[20 marks]**
▪ Task 01: LRU Page Replacement Algorithm	[10 marks]
▪ Task 02: Stack using Queue	[10 marks]

References and Additional Material:

Queue Data Structure

https://www.geeksforgeeks.org/queue-data-structure/