Data Structure Algorithms & Applications (CT-159) Lab 05

Queue, Circular Queue, Double Ended Queue

Objectives

The objective of the lab is to get students familiar with Queue data structure, its variants, and applications.

Tools Required

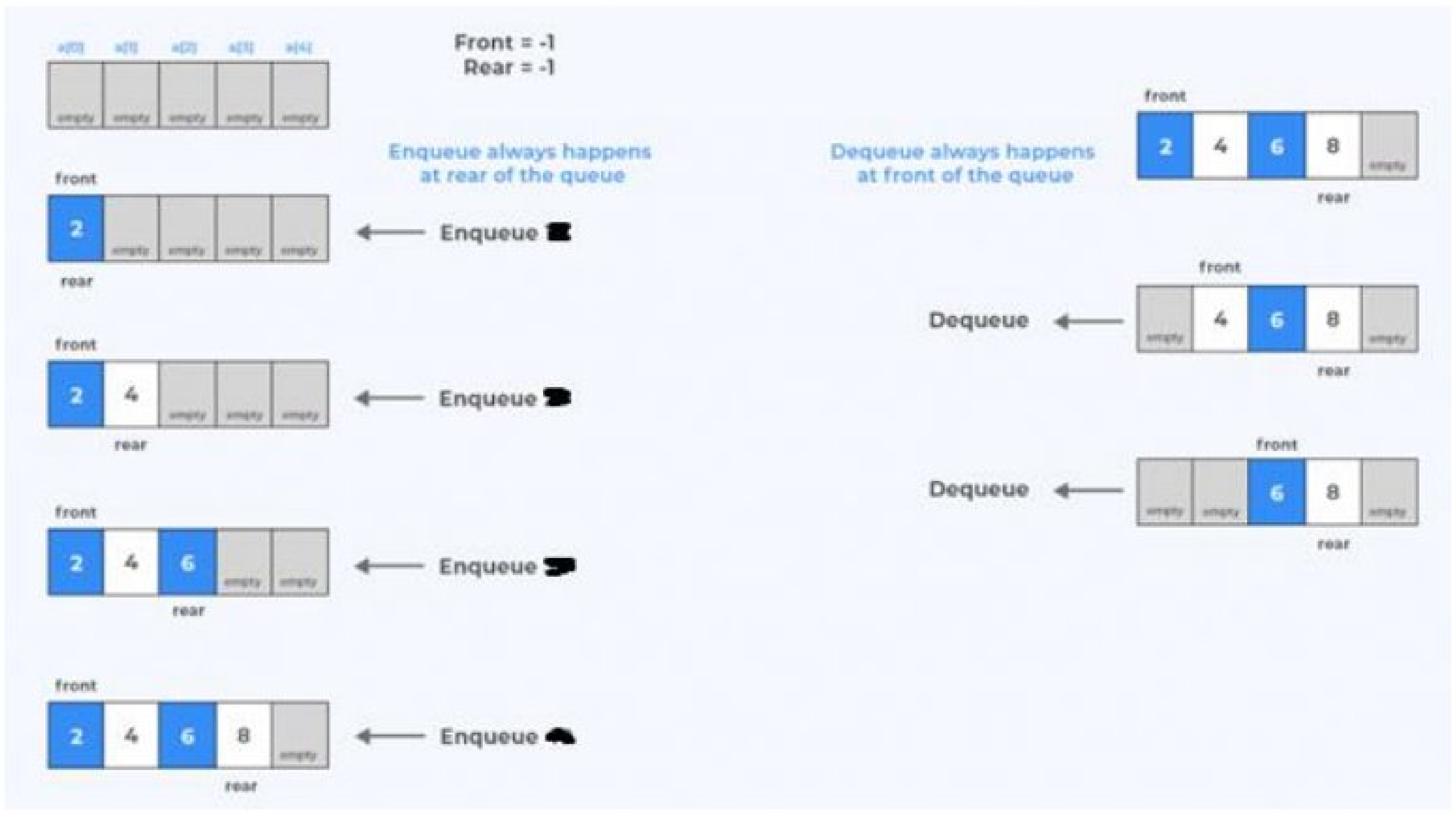
Dev C++ IDE

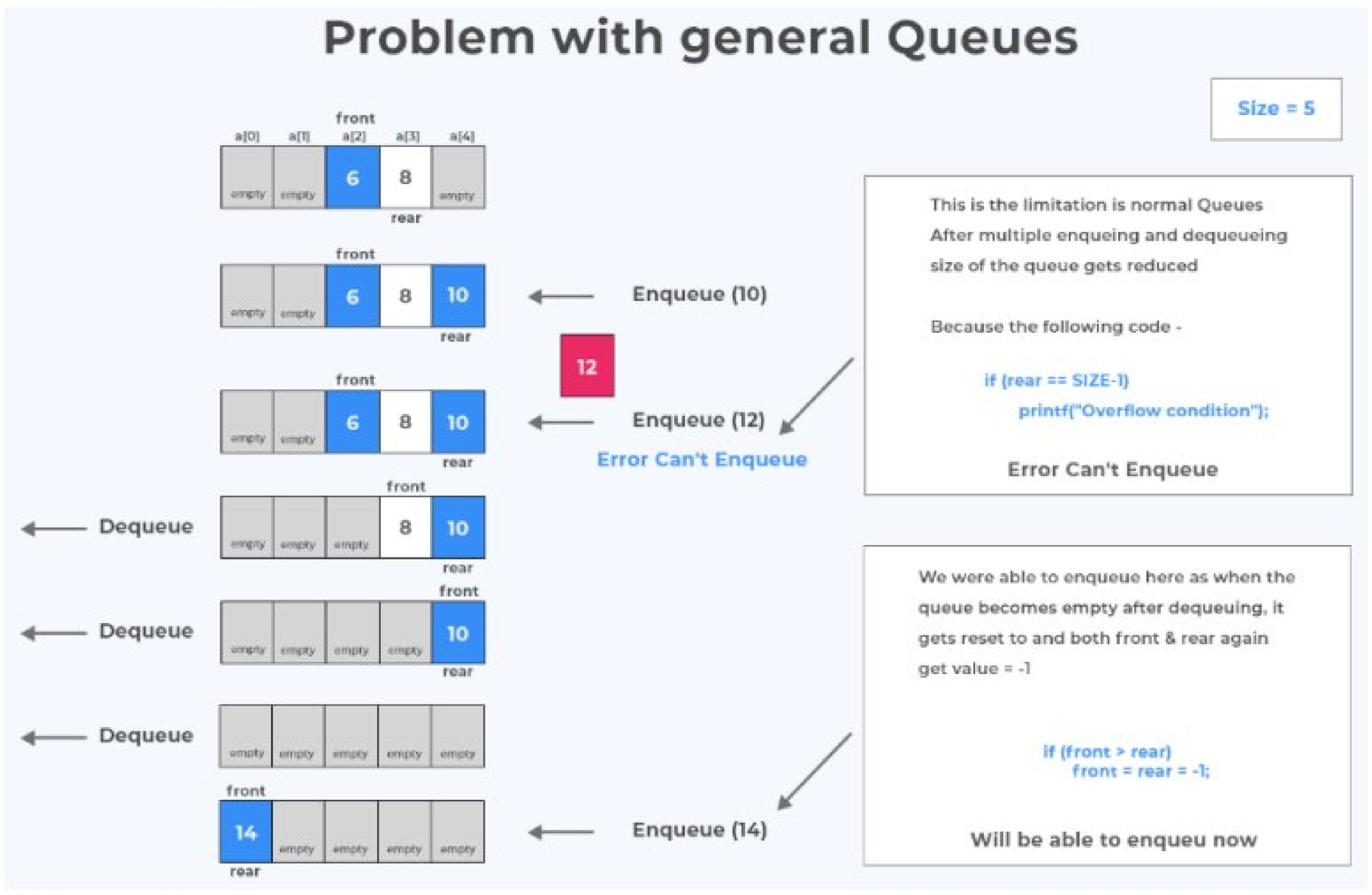
Course Coordinator –
Course Instructor –
Lab Instructor –
Prepared By Department of Computer Science and Information Technology
NED University of Engineering and Technology

Introduction

A queue is also a linear data structure which works according to FIFO (First IN First Out) manner. Of the two ends of the queue, one is designated as the front – where elements are extracted (operation called dequeue), and another is the rear, where elements are inserted (operation called enqueue). The following operations are needed to properly manage a queue:

- clear()—Clear the queue.
- isEmpty()—Check to see if the queue is empty.
- enqueue(el)—Put the element el at the end of the queue.
- dequeue()—Take the first element from the queue.
- firstEl()—Return the first element in the queue without removing it
- lastEl()—Return the last element in the queue without removing it


Front = -1
Rear = -1
Enqueue always happens at rear of the queue
Dequeue always happens at front of the queue
front
rear
Enqueue
Dequeue



This is the limitation is normal Queues
After multiple enqueing and dequeueing
size of the queue gets reduced

Because the following code -

```
if (rear == SIZE-1)
    printf("Overflow condition");
```

Error Can't Enqueue

We were able to enqueue here as when the queue becomes empty after dequeuing, it gets reset to and both front & rear again get value = -1

```
if (front > rear)
    front = rear = -1;
```

Will be able to enqueu now

Advantages of Linear Queue:

- **Simple Implementation**: Linear queues are easy to implement and understand, making them suitable for simple use cases where the data structure does not require dynamic resizing.

Limitations of Linear Queue:

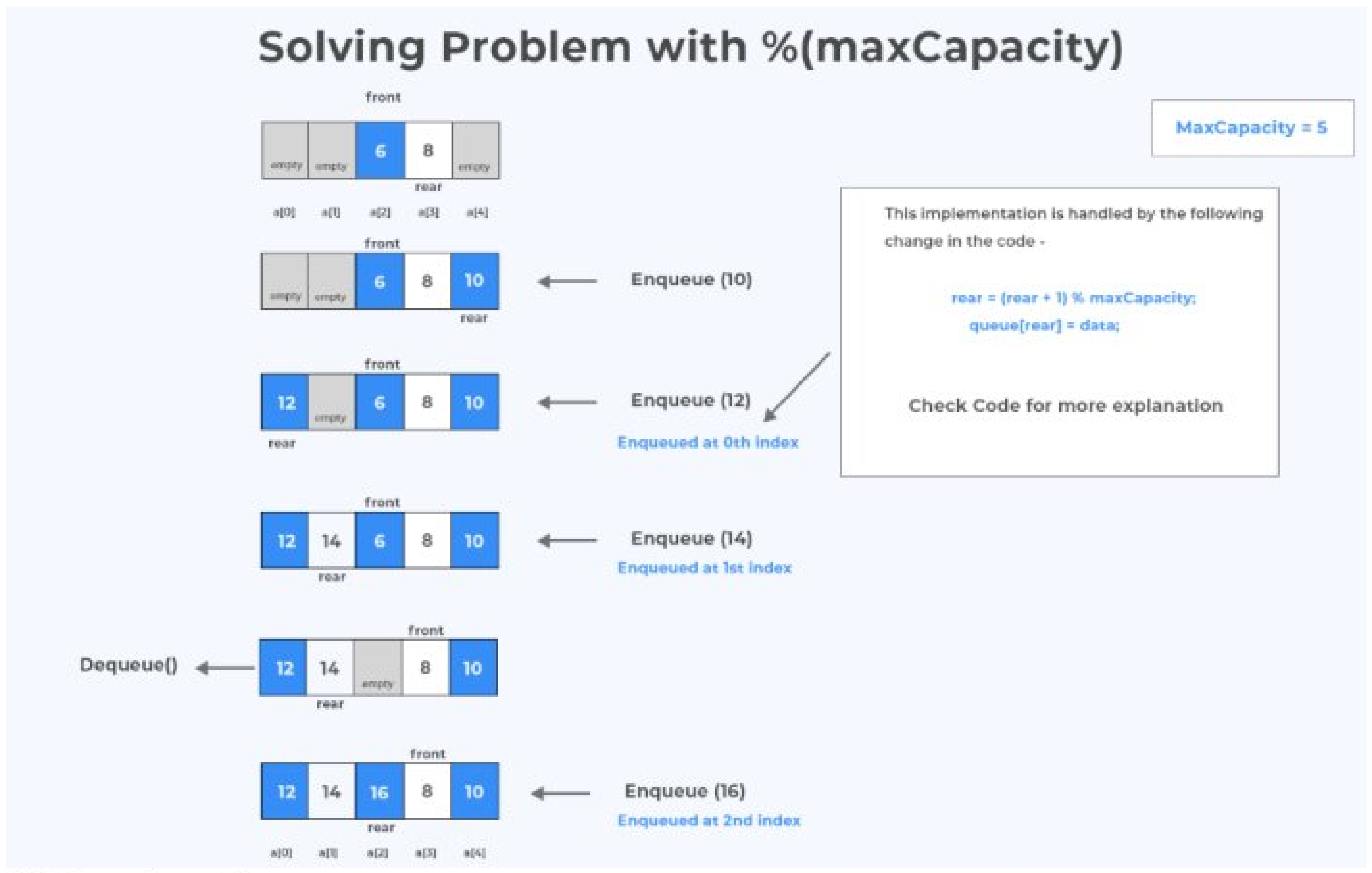
- **Wasted Space**: When elements are dequeued from the front, the space they occupied becomes unusable unless the remaining elements are shifted. This shifting process can be inefficient.
- **Fixed Size**: The size of the array used to implement the queue is fixed, meaning once the maximum capacity is reached, no more elements can be enqueued without resizing the array or creating a new queue.

Example 01: Implementation of Generic Linear Queue Data Structure using Array

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

template <typename T>
class Queue {
private:
  T* data;                // Array to store queue elements
  size_t capacity;          // Maximum number of elements the queue can hold
  size_t front;           // Index of the front element
  size_t rear;           // Index of the next position to insert an element

public:
  Queue(size_t initialCapacity = 4) : capacity(initialCapacity), front(-1), rear(-1) {
    data = new T[capacity];                 // Dynamically allocate array for queue elements
  }

  ~Queue() {                        // Destructor to clean up allocated memory
    delete[] data;    }

  void enqueue(const T& value) {               // Add an element to the rear of the queue
    if (rear == capacity-1) {
      throw overflow_error("Queue is full");   }
    data[++rear] = value;                 // Insert the element at the rear
  }

  void dequeue() {                  // Remove the front element from the queue
    if (isEmpty()) {
      throw out_of_range("Queue is empty");
     }
    front++;                        // Increment front index
    if(front>rear){
      front=rear=-1;
    }
  }

  T& peek() const {                 // Get the front element of the queue without removing it
    if (isEmpty()) {
      throw out_of_range("Queue is empty");    }
    return data[front+1];              // Return the front element
  }

  bool isEmpty() const {           // Check if the queue is empty
    return rear == -1;
  }

  size_t getSize() const {          // Get the current size of the queue
    return rear;
  }
};

int main() {
  Queue<int> q;
  q.enqueue(1);
  q.enqueue(2);
  q.enqueue(3);
  q.enqueue(4);
  cout << "Front element: " << q.peek() << std::endl;
  q.dequeue();

  cout << "Front element after dequeue: " << q.peek() << std::endl;
  cout << "Queue current size: " << q.getSize() << std::endl;
  return 0;
}
```

Circular Queue

A **circular queue** is a linear data structure that uses a fixed-size array in a circular fashion. In a circular queue, the last position is connected back to the first position to make a circle. This setup allows efficient use of the available space, enabling insertion and deletion operations to be performed in constant time, O(1), if the queue is not full or empty.



Advantages of Circular Queue

- **No Wasted Space**: Unlike linear queues, circular queues make full use of the array's space by treating it as a circular structure. This avoids the problem of wasted space due to shifting elements after a dequeue operation, as seen in linear queues.
- **O(1) Time Complexity**: Both enqueue and dequeue operations can be performed in constant time, O(1), regardless of the number of elements in the queue. This makes circular queues highly efficient for applications that require frequent insertion and removal of elements.

Disadvantages of Circular Queue

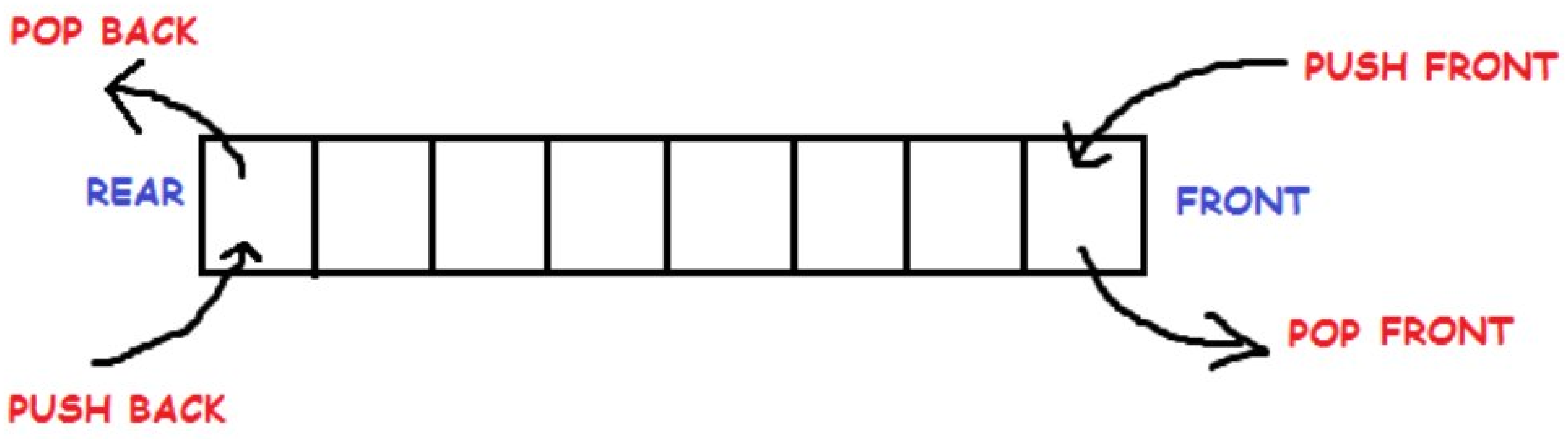
- **Fixed Capacity**: If circular queues are implemented using a fixed-size array, the maximum number of elements they can hold is predefined. This means the queue can become full, at which point no more elements can be added without resizing or replacing the queue.
- **Additional Logic for Wrap-Around**: The circular nature requires additional logic to handle the wrap-around of the front and rear pointers. This can make the implementation slightly more complex than a linear queue, especially in managing conditions like queue full or empty.

Example 02: Implementation of Generic Circular Queue Data Structure using Array

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

template <typename T>
class Queue {
private:
  T* data;
  size_t capacity;
  size_t front;
  size_t rear;
  size_t size;

  void resize() {
    size_t newCapacity = capacity * 2;
    T* newData = new T[newCapacity];

    for (size_t i = 0; i < size; ++i) {
      newData[i] = data[(front + i) % capacity];
    }

    delete[] data;
    data = newData;
    capacity = newCapacity;
    front = 0;
    rear = size;
  }

public:
  Queue(size_t initialCapacity = 4) : capacity(initialCapacity), front(0), rear(0), size(0) {
        data = new T[capacity];
    }

    ~Queue() {
        delete[] data;
    }

    void enqueue(const T& value) {
        if (size == capacity) {
            resize();
        }
        data[rear] = value;
        rear = (rear + 1) % capacity;
        ++size;
    }

    void dequeue() {
        if (isEmpty()) {
            throw out_of_range("Queue is empty");
        }
        front = (front + 1) % capacity;
        --size;
    }

    T& peek() const {
        if (isEmpty()) {
            throw out_of_range("Queue is empty");
        }
        return data[front];
    }

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

    size_t getSize() const {
        return size;
    }
};
int main() {
    Queue<int> q;
    q.enqueue(1);
    q.enqueue(2);
    q.enqueue(3);

    cout << "Front element: " << q.peek() << endl;
    q.dequeue();
    cout << "Front element after dequeue: " << q.peek() << endl;
    cout << "Queue size: " << q.getSize() << endl;
    return 0;
}
```

Double Ended Queue

A double-ended queue (commonly referred to as a deque, pronounced "deck") is a linear data structure that allows insertion and deletion of elements from both ends — the front and the rear. This makes the deque a versatile data structure that can act both as a queue (FIFO) and a stack (LIFO). Managing indices for both ends can make the implementation more complex compared to simpler data structures like stacks or linear queues. Care must be taken to handle edge cases such as deque underflow or overflow.



Applications

1. Print Queue in Operating Systems

In an office setting with multiple users sharing several printers, efficient management of print jobs is crucial. To avoid conflicts and ensure that print jobs are handled in the order they are received, a print queue is implemented.

How It Works:

- **Print Spooler**: A print spooler uses a queue to manage print jobs sent to the printer. When a user submits a print job, the job is placed at the rear of the queue.
- **First In, First Out (FIFO)**: The printer processes jobs in the order they are added to the queue. This FIFO behavior ensures fairness so that jobs are printed in the sequence they are received.

Benefits:

- **Resource Optimization**: Prevents printers from being overloaded with multiple simultaneous requests.
- **User Satisfaction**: Ensures a predictable and fair order of print jobs, reducing user frustration and potential conflicts over printer use.

2. Task Scheduling in Operating Systems

In a multi-user operating system, multiple processes need to share the CPU efficiently. The operating system must ensure that each process gets a fair amount of CPU time to function correctly.

How It Works:

- **Round Robin Scheduling**: One common CPU scheduling algorithm is round robin, where each process is assigned a fixed time slice. The processes are stored in a queue.
- **Process Management**: When a process's time slice expires, it is removed from the front of the queue and placed at the rear, while the next process is moved to the front.
- **Continuous Execution**: This cycle continues, ensuring that all processes get regular CPU time and preventing any single process from monopolizing the CPU.

Benefits:

- **Fairness**: Ensures that all processes receive equal CPU time, preventing any one process from starving others.
- **Responsiveness**: Keeps the system responsive by regularly switching between processes, which is especially important in interactive systems.

3. Customer Service Systems

A call center needs to manage incoming customer calls efficiently to reduce wait times and improve customer satisfaction.

How It Works:

- **Call Queue**: Incoming calls are placed in a queue. Each new call is added to the rear of the queue, and calls are answered in the order they are received.
- **Load Balancing**: The system may use multiple queues to balance loads among available agents.

Calls are dequeued and assigned to agents as they become available.

- **Priority Handling**: Some systems use priority queues, where certain calls (e.g., VIP customers) may be moved ahead in the queue based on predefined rules.

Benefits:

- **Efficient Call Handling**: Ensures that calls are handled in an orderly fashion, reducing the chances of call drops due to long wait times.
- **Improved Customer Satisfaction**: By managing calls effectively, customer wait times are minimized, leading to a better overall experience.

4. Web Server Request Management

Web servers need to handle multiple incoming requests from users browsing the internet. Efficient management of these requests ensures a smooth browsing experience.

How It Works:

- **Request Queue**: When a user sends an HTTP request to a web server, the request is placed in a queue.
- **Thread Pool**: A pool of worker threads dequeues requests from the queue and processes them.
- **Load Management**: If the server is too busy, incoming requests might be queued until a thread becomes available to handle them.

Benefits:

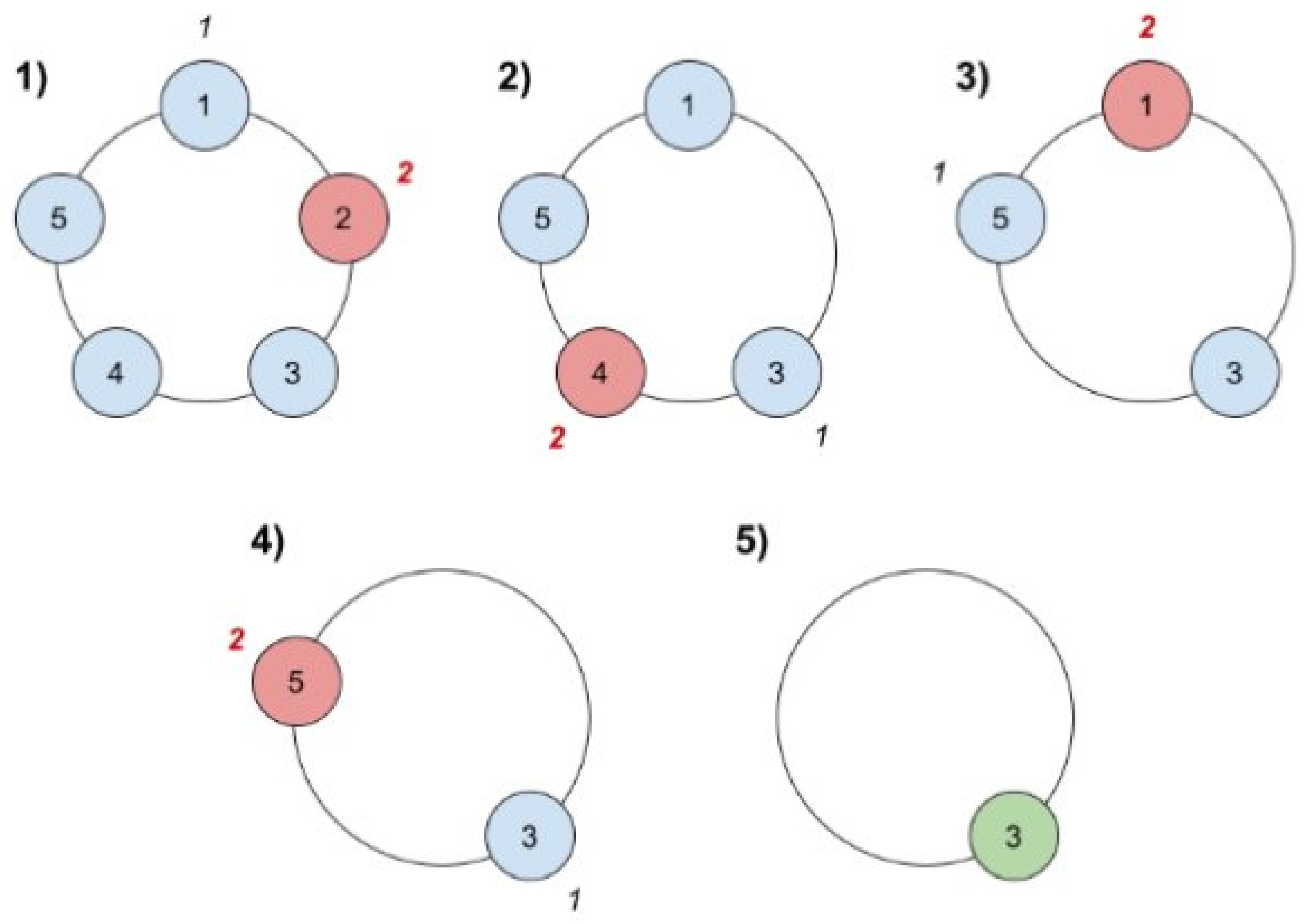
- **Scalability**: Helps in managing high volumes of requests efficiently without overloading the server.
- **Resource Optimization**: Ensures that server resources are used optimally, balancing between serving current requests and managing new ones.

Exercise

1. Design your implementation of the circular double-ended queue (deque). Implement the MyCircularDeque class:
   - MyCircularDeque(int k) Initializes the deque with a maximum size of k.
   - boolean insertFront() Adds an item at the front of Deque. Returns true if the operation is successful, or false otherwise.
   - boolean insertLast() Adds an item at the rear of Deque. Returns true if the operation is successful, or false otherwise.
   - boolean deleteFront() Deletes an item from the front of Deque. Returns true if the operation is successful, or false otherwise.
   - boolean deleteLast() Deletes an item from the rear of Deque. Returns true if the operation is successful, or false otherwise.
   - int getFront() Returns the front item from the Deque. Returns -1 if the deque is empty.
   - int getRear() Returns the last item from Deque. Returns -1 if the deque is empty.
   - boolean isEmpty() Returns true if the deque is empty, or false otherwise.
   - boolean isFull() Returns true if the deque is full, or false otherwise.
   - Main Program:

```
Int main( ){
MyCircularDeque myCircularDeque = new MyCircularDeque(3);
myCircularDeque.insertLast(1);  // return True
myCircularDeque.insertLast(2);  // return True
myCircularDeque.insertFront(3); // return True
myCircularDeque.insertFront(4); // return False, the queue is full.
myCircularDeque.getRear();     // return 2
myCircularDeque.isFull();     // return True
myCircularDeque.deleteLast();  // return True
myCircularDeque.insertFront(4); // return True
myCircularDeque.getFront();    // return 4 }
```

2. There are n friends that are playing a game. The friends are sitting in a circle and are numbered from 1 to n in clockwise order. More formally, moving clockwise from the ith friend brings you to the (i+1)th friend for 1 <= i < n, and moving clockwise from the nth friend brings you to the 1st friend. The rules of the game are as follows:
   - Start at the 1st friend.
   - Count the next k friends in the clockwise direction including the friend you started at. The counting wraps around the circle and may count some friends more than once.
   - The last friend you counted leaves the circle and loses the game.
   - If there is still more than one friend in the circle, go back to step 2 starting from the friend immediately clockwise of the friend who just lost and repeat.
   - Else, the last friend in the circle wins the game.
   - Given the number of friends, n, and an integer k, return the winner of the game.



```
Input: n = 5, k = 2
Output: 3
```

3. You are designing a simple customer service call center simulation. The call center has a fixed number of customer service representatives (CSRs), and it needs to manage incoming calls efficiently. You will implement a call queue system where calls are handled on a first-come, first-served basis.

Requirements:

- Queue for Incoming Calls: Use a queue to manage incoming customer calls.
- Fixed Number of CSRs: The call center has a fixed number of CSRs. Each CSR can handle one call at a time.

Instructions:

- Define the class for a call with attributes like Call ID, Arrival Time, and Customer Name, and a constructor to initialize these attributes.
- Define another class for simulating a call center
- Define a queue as a data member to hold incoming calls, an int CSR for defining the number of CSRs and a bool vector for defining whether a CSR is available or not
- Create a function to add a new call to the queue.
- Create a function to process calls, assigning them to available CSRs.
- Simulate the processing time of each call and update the status of the call once completed.
- Initialize the call center with a fixed number of CSRs.
- Simulate the arrival of calls at different times.
- Process calls based on CSR availability and handle calls in a first-come, first-served manner.

4. Design an algorithm that accepts a stream of integers and retrieves the product of the last k integers of the stream. Implement the ProductOfNumbers class:
   - ProductOfNumbers() Initializes the object with an empty stream.
   - void add(int num) Appends the integer num to the stream.
   - int getProduct(int k) Returns the product of the last k numbers in the current list. You can assume that always the current list has at least k numbers.
   - The test cases are generated so that, at any time, the product of any contiguous sequence of numbers will fit into a single 32-bit integer without overflowing.
   - Main Program

```
Int main(){
ProductOfNumbers productOfNumbers = new ProductOfNumbers();
productOfNumbers.add(3);     // [3]
productOfNumbers.add(0);     // [3,0]
productOfNumbers.add(2);     // [3,0,2]
productOfNumbers.add(5);     // [3,0,2,5]
productOfNumbers.add(4);     // [3,0,2,5,4]
productOfNumbers.getProduct(2); // return 20. The product of the last 2 numbers is 5 * 4 = 20
productOfNumbers.getProduct(3); // return 40. The product of the last 3 numbers is 2 * 5 * 4 = 40
productOfNumbers.getProduct(4); // return 0. The product of the last 4 numbers is 0 * 2 * 5 * 4 = 0
productOfNumbers.add(8);      // [3,0,2,5,4,8]
productOfNumbers.getProduct(2); // return 32. The product of the last 2 numbers is 4 * 8 = 32}
```

5. For a stream of integers, implement a data structure that checks if the last k integers parsed in the stream are equal to value. Implement the DataStream class:
   - DataStream(int value, int k) Initializes the object with an empty integer stream and the two integers value and k.
   - boolean consec(int num) Adds num to the stream of integers. Returns true if the last k integers are equal to value, and false otherwise. If there are less than k integers, the condition does not hold true, so returns false.
   - Main Program:

```
void main(){
DataStream dataStream = new DataStream(4, 3); //value = 4, k = 3
dataStream.consec(4); // Only 1 integer is parsed, so returns False.
dataStream.consec(4); // Only 2 integers are parsed.
                // Since 2 is less than k, returns False.
dataStream.consec(4); // The 3 integers parsed are all equal to value, so returns True.
dataStream.consec(3); // The last k integers parsed in the stream are [4,4,3].
                // Since 3 is not equal to value, it returns False.
}
```

Lab 03 Evaluation		
Student Name:	**Student ID:**	**Date:**
Rubric	**Marks (25)**	**Remarks by teacher in accordance with the rubrics**
R1		
R2		
R3		
R4		
R5		