TRIBHUVAN UNIVERSITY

Paschimanchal Campus

Lamachaur 16, Pokhara

Department of Computer Engineering

Lab Report On:

Lab No: 4

Title:

Submitted By

Name: Anish Karki

Roll No: PAS080BCT007

Submitted To: HPB Sir

Date of submission: 03/07/2025

PRACTICAL-1
CIRCULAR QUEUE

THEORY

A circular queue is a linear data structure that follows the First-In-First-Out (FIFO) principle, but unlike a simple linear queue, it connects the last position back to the first position to form a circle. This is done to efficiently utilize the memory by avoiding the wastage of space that occurs in a linear queue after repeated dequeue operations.

In a linear queue, once the rear reaches the last index, no more elements can be inserted even if there is free space at the front due to previous deletions. A circular queue solves this problem by treating the array as circular and allowing the rear to wrap around to the front when there's space.

Time Complexity:

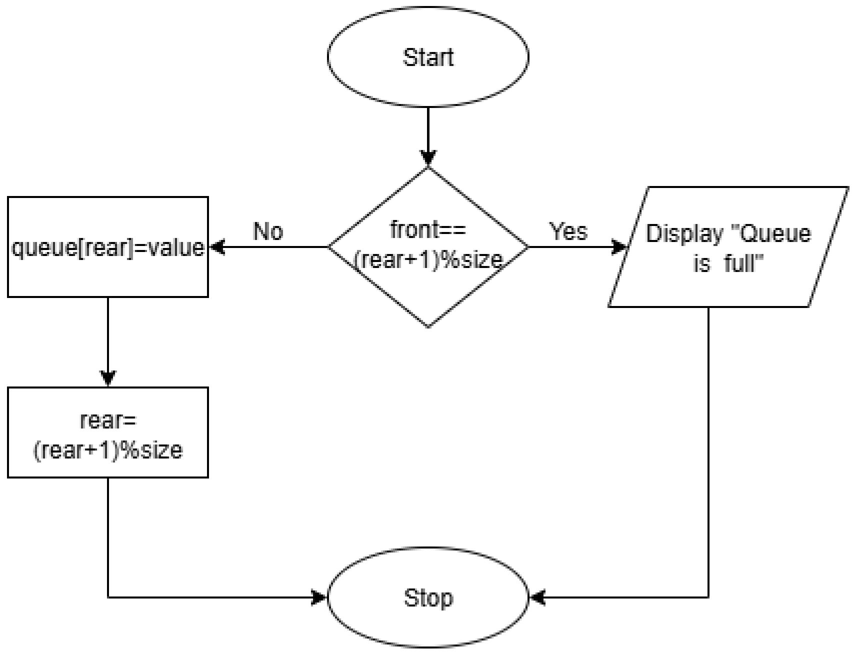
Enqueue: O(1)
Dequeue: O(1)

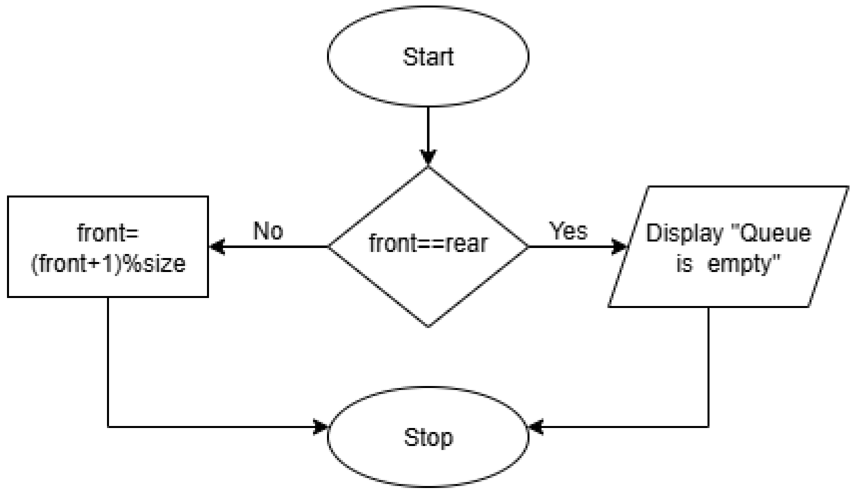
This circular queue implementation has some limitations, such as lacking a destructor to free memory and not providing a way to display or retrieve elements when dequeuing.

ALGORITHM

1) Start
2) Initialize front = 0, rear = 0, and create an array queue of given size.
3) Enqueue Operation:
    - If front == (rear + 1) % size, print "Queue is full".
    - Else, set queue[rear] = value and update rear = (rear + 1) % size.
4) Dequeue Operation:
    - If front == rear, print "Queue is empty".
    - Else, update front = (front + 1) % size.
5) Stop

FLOWCHART


Start
front==
(rear+1)%size
No
Yes
queue[rear]=value
Display "Queue
is full"
rear=
(rear+1)%size
Stop


Start
front==rear
No
Yes
front=
(front+1)%size
Display "Queue
is empty"
Stop

CODE:

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

class CircularQueue {
public:
    int front, rear;
    int *queue;
    int size;

    CircularQueue(int size) : queue(new int[size]), front(0), rear(0), size(size) {}

    void enqueue(int x) {
        if (front == ((rear + 1) % size))
            cout << "Queue is full" << endl;
        else {
            queue[rear] = x;
            rear = (rear + 1) % size;
        }
    }

    void dequeue() {
        if (front == rear)
            cout << "Queue is empty" << endl;
        else
            front = (front + 1) % size;
    }
};

int main() {
    CircularQueue q(5);
    q.enqueue(1);
    q.enqueue(2);
    q.dequeue();
    q.enqueue(3);
    q.dequeue();
    q.dequeue();
    return 0;
}
```

PRACTICAL-2
CREATION OF SINGLY LINKED LIST

THEORY

A linked list is a linear data structure used to store a collection of elements, called nodes, where each node is connected to the next using a pointer. In a singly linked list, each node contains two parts:

Data – the actual value stored

Next pointer – a reference to the next node in the list

Unlike arrays, linked lists do not require elements to be stored in contiguous memory locations. Instead, each node is dynamically allocated and connected in sequence through pointers. The first node is pointed to by a special pointer called the head, and the last node's next pointer is set to NULL to indicate the end of the list.

Singly linked lists are used in scenarios where the size of data is unknown at compile time or when frequent insertions and deletions are needed. This structure allows dynamic memory usage, meaning memory is allocated and deallocated as needed, improving flexibility and reducing memory wastage.

ALGORITHM

Steps:

1.Start

2.Define a Node class:

- Contains an integer data.
- Contains a pointer next to the next node.
- Constructor initializes data and sets next to NULL.

3.Define a LinkedList class:

- Contains a pointer head to the first node.
- Constructor initializes head to NULL.

4.In the main function:

- Create an object list of the LinkedList class.

5.End

CODE:

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

class Node {
public:
  int data;
  Node* next;

  Node(int data) {
    this->data = data;
    this->next = NULL;
  }
};

class LinkedList {
  Node* head;

public:
  LinkedList() : head(NULL) {}
};

int main() {
  LinkedList list;
  return
```

CONCLUSION

In this lab, we successfully implemented the creation of a singly linked list and performed basic queue operations such as enqueue and dequeue. The singly linked list allowed dynamic memory allocation and efficient insertion of elements. The enqueue operation added elements to the rear of the queue, while the dequeue operation removed elements from the front, maintaining the First-In-First-Out (FIFO) principle. This practical exercise deepened our understanding of linked data structures and their use in queue implementation.