## Task 1

#pragma once

#include <iostream>

using namespace std;

template <class T>

class List {

protected:

T\* arr;

int maxSize, currentSize;

public:

List();

List(const List&);

virtual void addElement(T) = 0;

virtual T removeElement() = 0;

void regrow();

virtual ~List();

};

template<class T>

List<T>::List() : maxSize(1), currentSize(0) {

arr = new T[maxSize];

}

template<class T>

List<T>::List(const List& obj) : currentSize(obj.currentSize), maxSize(obj.maxSize) {

arr = new T[maxSize];

for (int a = 0; a < currentSize; a++) {

arr[a] = obj.arr[a];

}

}

template<class T>

void List<T>::regrow() {

maxSize \*= 2; // Double the size to reduce frequent resizing

T\* newArr = new T[maxSize];

for (int a = 0; a < currentSize; a++) {

newArr[a] = arr[a];

}

delete[] arr;

arr = newArr;

}

template<class T>

List<T>::~List() {

delete[] arr;

}

## Task 2 & 3:-

#pragma once

#include "List.h"

template<class T>

class CircularQueue : public List<T> {

int front, rear;

public:

CircularQueue();

CircularQueue(const CircularQueue&);

void addElement(T) override;

T removeElement() override;

bool empty() const;

bool full() const;

int size() const;

T getFront() const;

void enqueue(T);

T dequeue();

template<class Q>

friend void showQueue(const CircularQueue<Q>&);

~CircularQueue();

};

template<class T>

CircularQueue<T>::CircularQueue() : List<T>(), front(-1), rear(-1) {}

template<class T>

CircularQueue<T>::CircularQueue(const CircularQueue& obj) : List<T>(obj), front(obj.front), rear(obj.rear) {}

template<class T>

bool CircularQueue<T>::empty() const {

return front == -1;

}

template<class T>

bool CircularQueue<T>::full() const {

return (rear + 1) % List<T>::maxSize == front;

}

template<class T>

void CircularQueue<T>::addElement(T value) {

if (full()) {

List<T>::regrow();

}

if (front == -1) {

front = 0;

}

rear = (rear + 1) % List<T>::maxSize;

List<T>::arr[rear] = value;

List<T>::currentSize++;

}

template<class T>

T CircularQueue<T>::removeElement() {

if (empty()) {

cout << "Queue is Empty!..." << endl;

return T(); // Return default value if empty

}

T temp = List<T>::arr[front];

if (front == rear) {

front = rear = -1;

}

else {

front = (front + 1) % List<T>::maxSize;

}

List<T>::currentSize--;

return temp;

}

template<class T>

int CircularQueue<T>::size() const {

return List<T>::currentSize;

}

template<class T>

T CircularQueue<T>::getFront() const {

if (empty()) {

cout << "Queue is Empty!" << endl;

return T(); // Return default value if empty

}

return List<T>::arr[front];

}

template<class T>

void CircularQueue<T>::enqueue(T v) {

addElement(v);

}

template<class T>

T CircularQueue<T>::dequeue() {

return removeElement();

}

template<class T>

CircularQueue<T>::~CircularQueue() {}

## OUTPUT SCREENSHOT:

