**Data Structures & Algorithms**

**ASSIGNMENT NO. 1**

A black background with blue text

Description automatically generated

**Name:** Saad Ahmad

**Enrollment No:** 01-134222-130

**Class:** BSCS - 3A

**Array:**

#include <iostream>

using namespace std;

class Queue {

int front;

int rear;

int count;

int\* arr;

int size;

public:

Queue(int s) {

front = rear = 0;

count = 0;

size = s;

arr = new int[size];

}

bool Full() {

if (count == size) {

return true;

}

else {

return false;

}

}

bool Empty() {

if (count == 0) {

return true;

}

else {

return false;

}

}

void EnqueueR(int val) {

if (Full()) {

cout << "Queue Overflow!" << endl;

}

else {

arr[rear] = val;

rear = (rear + 1) % size;

count++;

}

}

void EnqueueF(int val) {

if (Full()) {

cout << "Queue Overflow!" << endl;

}

else {

front = (front - 1 + size) % size;

arr[front] = val;

count++;

}

}

void DequeueF() {

if (Empty()) {

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

}

else {

front = (front + 1) % size;

count--;

}

}

void DequeueR() {

if (Empty()) {

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

}

else {

rear = (rear - 1 + size) % size;

count--;

}

}

int getFront() {

return arr[front];

}

void display() {

if (Empty()) {

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

}

else {

int f = front;

int c = count;

while (c > 0) {

cout << arr[f] << endl;

f = (f + 1) % size;

c--;

}

}

}

};

int main()

{

Queue q1(5);

q1.EnqueueR(10);

q1.EnqueueR(20);

q1.EnqueueR(30);

q1.EnqueueR(40);

q1.EnqueueF(5);

q1.DequeueF();

q1.EnqueueF(50);

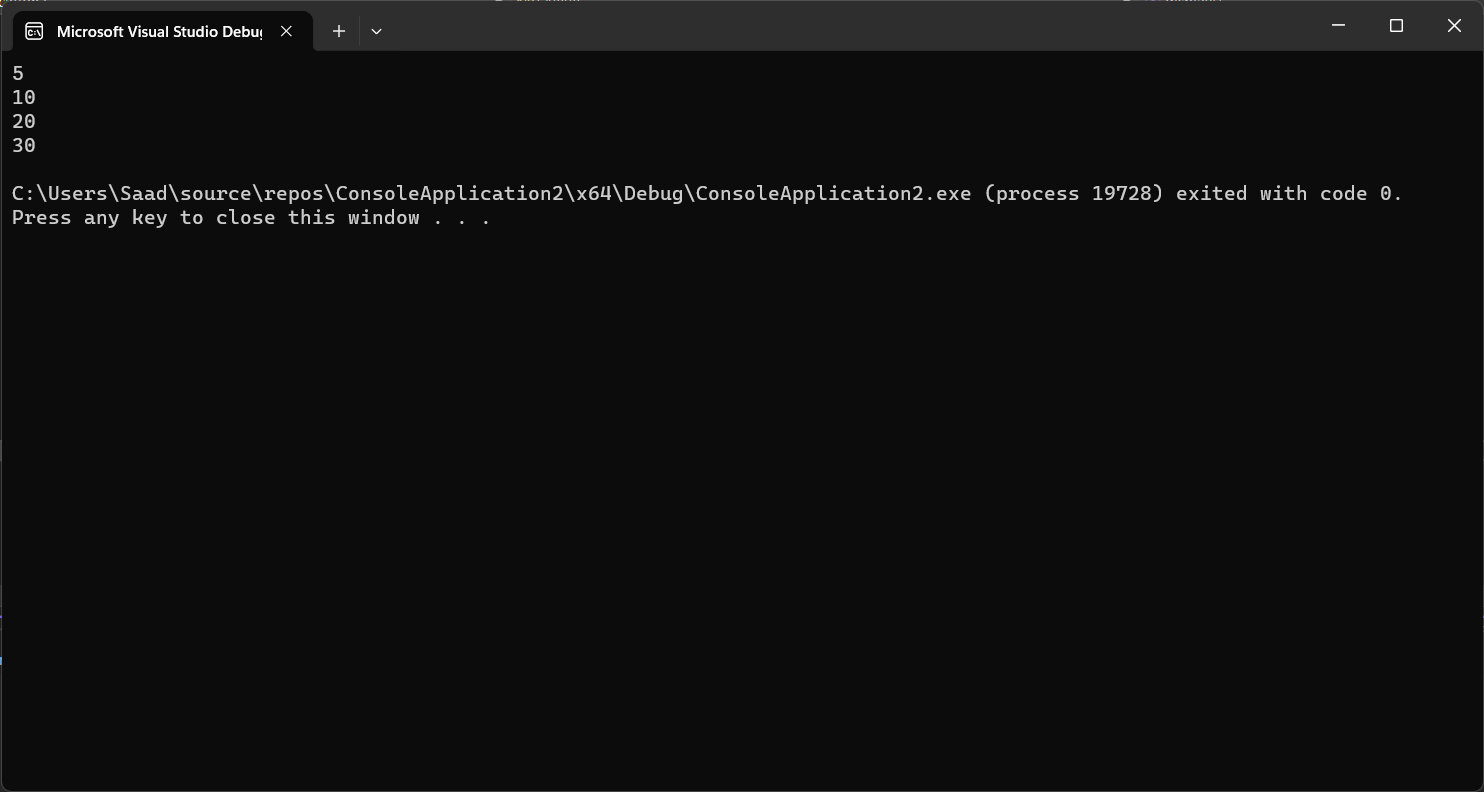
q1.DequeueF();

q1.EnqueueF(5);

q1.DequeueR();

q1.display();

}



**Explanation:**

This code implements a double ended circular queue. A double ended circular queue is a type of a queue that allows elements to be enqueued and dequeued from both the front and the rear. The Queue class includes essential data members such as front, rear, count, arr, and size. The front and rear variables keep track of the queue's front and rear positions, while count maintains the count of elements in the queue. The arr array dynamically stores the elements, and size specifies the maximum capacity. Functions are provided for enqueueing and dequeuing elements from both ends of the queue, checking whether the queue is full or empty, retrieving the front element, and displaying the queue's contents.

**Time complexity analysis:**

* EnqueueR and EnqueueF have a time complexity of O(1) because they involve simple array index manipulation.
* DequeueF and DequeueR also have a time complexity of O(1) for the same reason.
* The getFront method has a time complexity of O(1) since it directly accesses the front element.
* The Full and Empty methods have a time complexity of O(1) because they involve basic comparisons.

**LINKED LIST:**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

};

class Queue {

Node\* front;

Node\* rear;

public:

Queue() {

front = NULL;

rear = NULL;

}

bool Empty() {

if (front == NULL && rear == NULL) {

return true;

}

else {

return false;

}

}

void EnqueueR(int value) {

Node\* temp = new Node();

temp->data = value;

temp->next = NULL;

if (Empty()) {

temp->next = temp;

front = temp;

rear = temp;

}

else {

rear->next = temp;

temp->next = front;

rear = temp;

}

}

void EnqueueF(int value) {

Node\* temp = new Node();

temp->data = value;

temp->next = NULL;

if (Empty()) {

temp->next = temp;

front = temp;

rear = temp;

}

else {

temp->next = front;

front = temp;

rear->next = front;

}

}

void DequeueF() {

Node\* temp = front;

if (Empty()) {

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

return;

}

else if (front == rear) {

front = rear = NULL;

}

else {

front = front->next;

rear->next = front;

delete temp;

}

}

void DequeueR() {

Node\* temp = rear;

Node\* temp2 = front;

if (Empty()) {

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

return;

}

else if (front == rear) {

front = rear = NULL;

}

else {

while (temp2->next != temp) {

temp2 = temp2->next;

}

if (temp2->next == temp) {

rear = temp2;

rear->next = front;

delete temp;

}

}

}

int getFront() {

if (Empty()) {

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

return -1;

}

else {

int val = front->data;

return val;

}

}

void display() {

if (Empty()) {

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

return;

}

else {

Node\* temp = front;

while (!Empty()) {

cout << temp->data << endl;

temp = temp->next;

DequeueF();

}

}

}

};

int main()

{

Queue q1;

q1.EnqueueF(10);

q1.EnqueueF(20);

q1.EnqueueF(30);

q1.EnqueueR(5);

q1.EnqueueR(15);

q1.EnqueueR(25);

q1.DequeueF();

q1.DequeueR();

q1.display();

}

A screenshot of a computer

Description automatically generated

**Explanation:**  
This code presents a double ended circular queue implementation using a linked list. The Queue class maintains two pointers, front and rear, which indicate the front and rear of the circular queue. When empty, both pointers are set to NULL. The code provides functions to check if the queue is empty, enqueue elements at both the front and rear, and dequeue them from both ends. The display function prints the elements in the queue.

**Time Complexity Analysis:**

* EnqueueR and EnqueueF have a time complexity of O(1) because they involve creating a new node and updating pointers.
* DequeueF and DequeueR also have a time complexity of O(1) for the same reasons.
* The getFront method has a time complexity of O(1) since it directly accesses the front element.
* The Empty method has a time complexity of O(1) as it checks the pointers.

**Difference of Array based Circular queue and linked list based circular queue**

**Array-Based Circular Queue:** Array-based circular queues are memory-efficient and offer fast random access. However, they have a fixed size, and resizing can be complex and memory-intensive, leading to potential memory wastage.

**Linked List-Based Circular Queue:** Linked list-based circular queues dynamically adjust in size, making them memory-efficient for varying workloads. They are less memory-efficient due to node pointers, and their implementation is more complex. Random access is slower due to traversal.

**Practical examples of circular queues:**

**Task Scheduling:** Circular queues are used in scheduling algorithms in operating systems. Processes are placed in a circular queue and executed in a round-robin fashion, ensuring fair CPU time allocation for each process.

**Printing:** Printers use circular queues to manage print jobs. Print requests are enqueued and processed sequentially, ensuring a first-come, first-served order for printing.

**Cache Management:** Circular queues play a role in cache replacement policies. Cache entries are managed in a circular manner, ensuring optimal cache efficiency.

**Disk I/O:** File systems use circular queues for efficient disk I/O request management. When the disk is busy, requests are enqueued and processed when the disk becomes available.