

**National University of Computer and Emerging**

**Sciences**

**Chiniot-Faisalabad Campus BS (Artificial Intelligence)**

|  |  |
| --- | --- |
| **Name** | **M.Abdul Hanan** |
| **Reg.NO.** | **22F-3104** |
| **Section** | **BS(Ai)-3A1** |
| **Course** | **Data Structures** |
| **Department** | **CS Department** |
| **LAB** | **LAB Manual #6** |

**Task no 1:**

#include <iostream>

using namespace std;

class Queue {

public:

int front;

int rear;

int size;

int\* arr;

Queue() {

cout << "Enter size of Array :";

cin >> size;

front = -1;

rear = -1;

arr = new int[size];

for (int i = 0; i < size; i++) {

arr[i] = 0; // Initialize all elements to 0

}

}

// Check if the queue is full

bool Isfull() {

if (rear == size - 1) {

cout << "Queue is Full :";

return true;

}

else {

cout << "Queue is not Full :";

return false;

}

}

// Check if the queue is empty

bool IsEmpty() {

if (rear == -1 && front == -1) {

cout << "Queue is Empty:";

return true;

}

else {

cout << "Queue is not Empty :";

return false;

}

}

// Add an element to the rear of the queue

void Enqueue(int val) {

if (Isfull()) {

cout << "Queue is full" << endl;

return;

}

else if (IsEmpty()) {

front = rear = 0;

}

else {

rear++;

}

arr[rear] = val; // Add the value to the rear of the queue

}

int Dequeue() {

int x = 0;

if (IsEmpty()) {

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

return x;

}

else if (rear == front) {

x = arr[front];

arr[front] = 0;

front = rear = -1; // Reset front and rear for an empty queue

}

else {

x = arr[front];

arr[front] = 0;

front++;

}

return x; // Return the dequeued value

}

// Display the elements of the queue

void display() {

for (int i = 0; i < size; i++) {

cout << arr[i] << " ";

}

}

// Destructor to free dynamically allocated memory

~Queue() {

cout << "\nDestructor called :" << endl;

delete[] arr;

}

};

int main() {

int choice = 0;

Queue obj;

// Menu driven loop for interacting with the queue

do {

cout << "\nEnter 1 to check Array is Empty :";

cout << "\nEnter 2 to check Array is Full :";

cout << "\nEnter 3 to insert in Endqueue :";

cout << "\nEnter 4 to delete from dequeue :";

cout << "\nEnter 5 to Display the Array :";

cout << "\n\nEnter choice or press -1 to exit:";

cin >> choice;

if (choice == 1) {

obj.IsEmpty();

}

else if (choice == 2) {

obj.Isfull();

}

else if (choice == 3) {

int num;

cout << "\n Enter number :";

cin >> num;

obj.Enqueue(num);

}

else if (choice == 4) {

obj.Dequeue();

}

else if (choice == 5) {

obj.display();

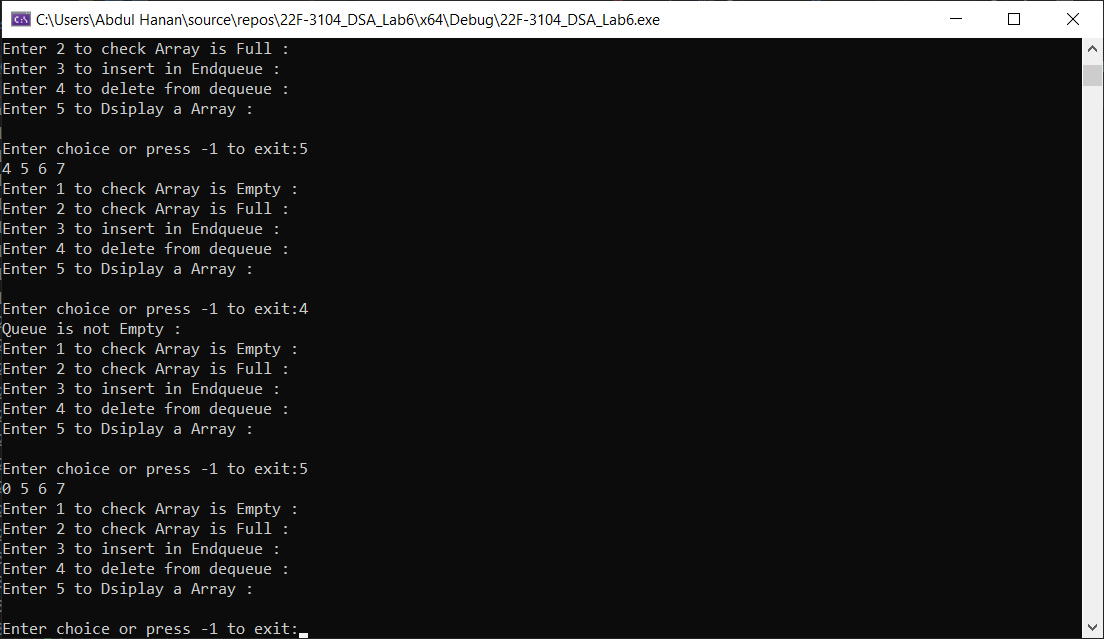
}

} while (choice != -1);

return 0;

}

**Screen Shot:**



**Task no 2:**

#include <iostream>

using namespace std;

class Queue {

public:

int front;

int rear;

int size;

int\* arr = new int[size]; // Dynamic memory allocation for the queue array

Queue() {

cout << "Enter size of Array :";

cin >> size;

front = -1;

rear = -1;

arr = new int[size];

for (int i = 0; i < size; i++) {

arr[i] = 0; // Initialize all elements of the queue array to 0

}

}

bool Isfull() {

if ((rear + 1) % size == front) { // Check if the queue is full

cout << "Queue is Full :";

return true;

}

else {

cout << "Queue is not Full :";

return false;

}

}

bool IsEmpty() {

if (rear == -1 && front == -1) { // Check if the queue is empty

cout << "Queue is Empty:";

return true;

}

else {

cout << "Queue is not Empty :";

return false;

}

}

void Enqueue(int val) {

if (Isfull()) {

cout << "Queue is full" << endl;

return;

}

else if (IsEmpty()) {

front = rear = 0; // Set front and rear to 0 when inserting the first element

}

else {

rear = (rear + 1) % size; // Increment rear circularly

}

arr[rear] = val; // Insert the value at the rear position

}

int Dequeue() {

int x = 0;

if (IsEmpty()) {

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

return x;

}

else if (rear == front) { // When there is only one element in the queue

x = arr[front];

arr[front] = 0;

front = rear = -1; // Reset front and rear

}

else {

x = arr[front];

arr[front] = 0;

front = (front + 1) % size; // Increment front circularly

}

return x;

}

void display() {

for (int i = 0; i < size; i++) {

cout << arr[i] << " "; // Display the elements of the queue

}

}

~Queue() {

cout << "\nDestructor called :" << endl;

delete[] arr; // Deallocate the dynamically allocated memory

}

};

int main() {

int choice = 0;

Queue obj;

do {

cout << "\nEnter 1 to check if the Array is Empty :";

cout << "\nEnter 2 to check if the Array is Full :";

cout << "\nEnter 3 to insert into the Queue :";

cout << "\nEnter 4 to delete from the Queue :";

cout << "\nEnter 5 to Display the Queue :";

cout << "\n\nEnter choice or press -1 to exit:";

cin >> choice;

if (choice == 1) {

obj.IsEmpty();

}

else if (choice == 2) {

obj.Isfull();

}

else if (choice == 3) {

int num;

cout << "\n Enter number :";

cin >> num;

obj.Enqueue(num);

}

else if (choice == 4) {

obj.Dequeue();

}

else if (choice == 5) {

obj.display();

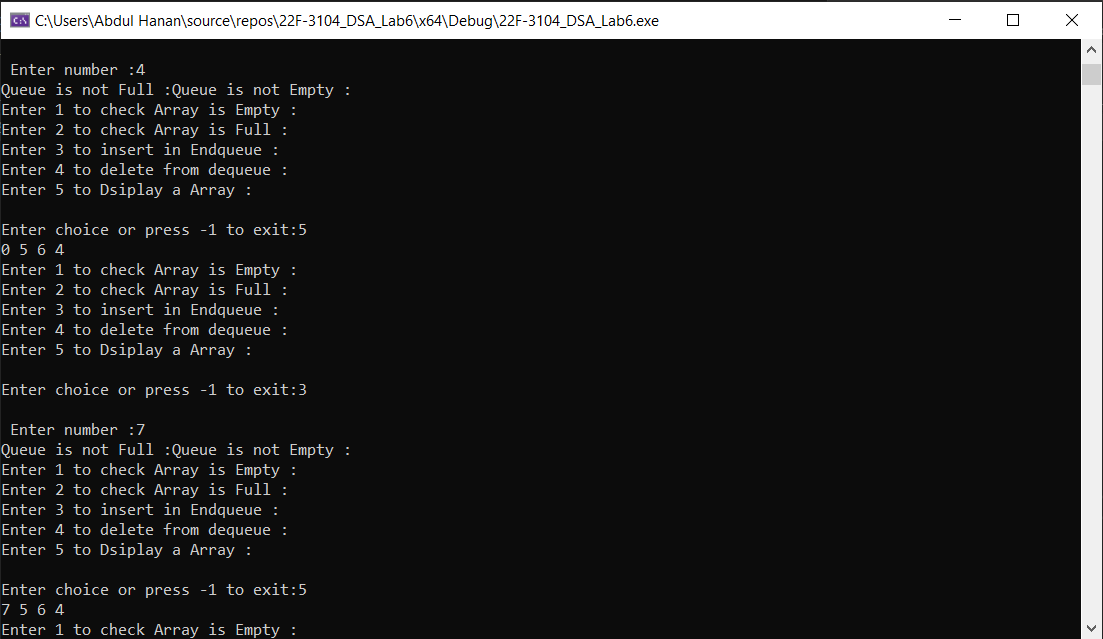
}

} while (choice != -1);

return 0;

}

**Screen Shot:**



**Task no 3:**

#include <iostream>

using namespace std;

class Queue {

public:

int data;

Queue\* next;

Queue(int val) {

data = val;

next = nullptr;

}

};

class QueueList {

private:

Queue\* front;

Queue\* rear;

public:

QueueList() {

front = rear = nullptr;

}

// Check if the Queue is empty

bool isEmpty() {

return front == nullptr;

}

// Enqueue add an element to the rear of the Queue

void enqueue(int val) {

Queue\* newNode = new Queue(val);

if (isEmpty()) {

// If the Queue is empty set both front and rear to the new node

front = rear = newNode;

}

else {

// Otherwise, append the new node to the rear and update rear pointer

rear->next = newNode;

rear = newNode;

}

}

// Dequeue remove an element from the front of the Queue

void dequeue() {

if (isEmpty()) {

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

return;

}

// Remove the front element update front pointer and deallocate memory

Queue\* temp = front;

front = front->next;

delete temp;

}

// Display the elements in the Queue

void display() {

if (isEmpty()) {

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

return;

}

Queue\* current = front;

while (current != nullptr) {

cout << current->data << " ";

current = current->next;

}

cout << endl;

}

// Destructor to free memory by dequeuing all elements

~QueueList() {

while (!isEmpty()) {

dequeue();

}

}

};

int main() {

int choice = 0;

QueueList queue;

// Menu driven loop for the queue

QueueList obj;

do {

cout << "\nEnter 1 to check Array is Empty :";

cout << "\nEnter 2 to insert in Endqueue :";

cout << "\nEnter 3 to delete from dequeue :";

cout << "\nEnter 4 to Display the Array :";

cout << "\n\nEnter choice or press -1 to exit:";

cin >> choice;

if (choice == 1) {

obj.isEmpty();

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

}

else if (choice == 2) {

int num;

cout << "\n Enter number :";

cin >> num;

obj.enqueue(num);

}

else if (choice == 3) {

obj.dequeue();

}

else if (choice == 4) {

obj.display();

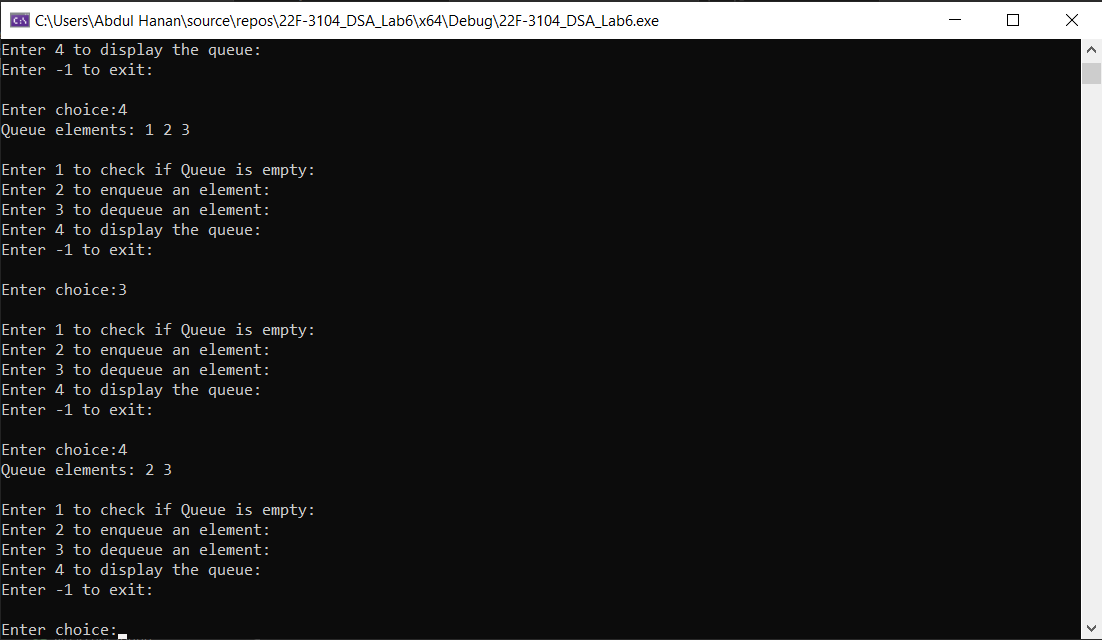
}

} while (choice != -1);

return 0;

}

**Screen Shot:**



**Task no 4:**

#include <iostream>

#include <windows.h>

using namespace std;

struct Process {

int processID;

int executionTime;

Process(){

processID = 1;

executionTime = 2;

}

Process(int id, int time)

{

processID = id;

executionTime = time;

}

};

int main() {

const int MAX\_PROCESSES = 100; // Maximum number of processes adjust as needed

Process processes[MAX\_PROCESSES];

int numProcesses = 0;

int processID, executionTime;

// Input processes until the user enters a sentinel value

while (true) {

cout << "Enter Process ID (or enter -1 to finish): ";

cin >> processID;

if (processID == -1) {

break;

}

cout << "Enter Execution Time in seconds for Process " << processID << ": ";

cin >> executionTime;

processes[numProcesses++] = Process(processID, executionTime);

}

int timeQuantum;

cout << "Enter the time quantum for Round Robin scheduling: ";

cin >> timeQuantum;

// Round Robin scheduling

int currentIndex = 0;

while (currentIndex < numProcesses) {

int remainingTime = processes[currentIndex].executionTime;

if (remainingTime <= timeQuantum) {

// Execute the process for its remaining time

cout << "Executing Process " << processes[currentIndex].processID << " for " << remainingTime << " seconds." << endl;

Sleep(remainingTime \* 1000); // Sleep in milliseconds

currentIndex++;

}

else {

// Execute the process for a time quantum

cout << "Executing Process " << processes[currentIndex].processID << " for " << timeQuantum << " seconds." << endl;

Sleep(timeQuantum \* 1000); // Sleep in milliseconds

processes[currentIndex].executionTime -= timeQuantum;

currentIndex = (currentIndex + 1) % numProcesses; // Move to the next process in a circular manner

}

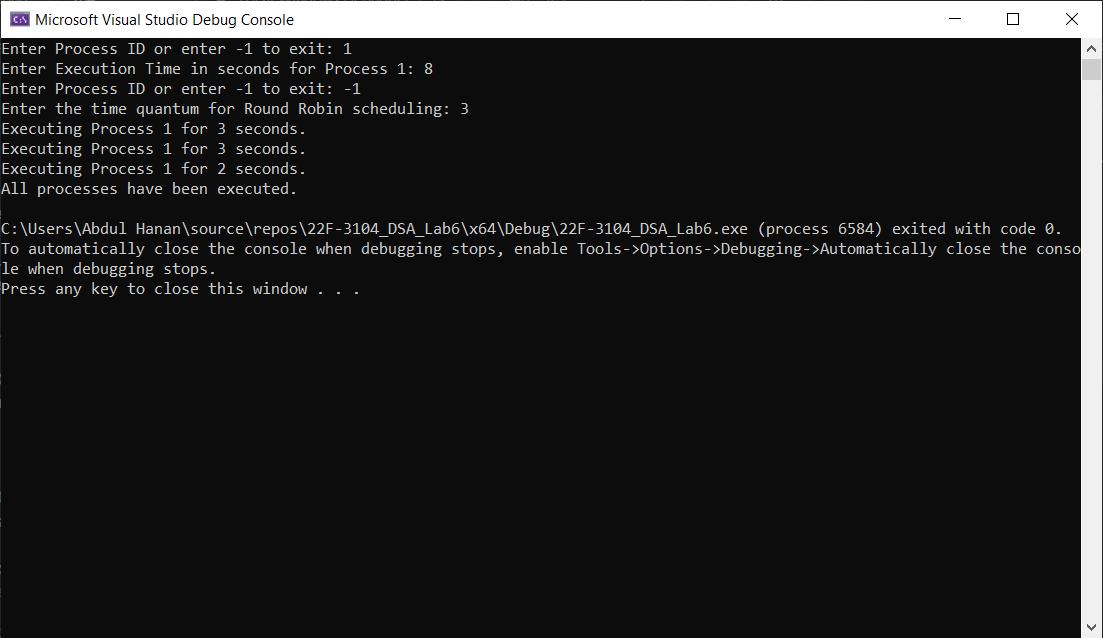
}

cout << "All processes have been executed." << endl;

return 0;

}

**Screen Shot:**

****

**Task no 5:**

#include <iostream>

using namespace std;

class Queue {

public:

int rear;

int front;

int size;

int\* items;

Queue(int Size) {

front = 0;

rear = -1;

size = Size;

items = new int[size];

}

// Check if the queue is empty

bool isEmpty() {

if (front > rear) {

return true;

}

else {

return false;

}

}

// Check if the queue is full

bool isFull() {

if (rear == size - 1) {

return true;

}

else {

return false;

}

}

// Add an element to the rear of the queue

void enqueue(int data) {

if (front > rear) {

cout << "Queue is full.\n";

return;

}

rear++;

items[rear] = data;

}

// Remove and return an element from the front of the queue

int dequeue() {

if (front > rear) {

return 0;

}

front++;

return items[front];

}

// Destructor to free the memory allocated for the queue

~Queue() {

delete[] items;

}

};

// Function to maximize profit by filling seats

int maximizeProfit(int\* seats, int num\_people, int rows) {

Queue obj(num\_people);

// Enqueue people in the queue

for (int i = 0; i < num\_people; i++) {

obj.enqueue(i);

}

int profit = 0;

// Continue processing while the queue is not empty

while (!obj.isEmpty()) {

int current = obj.dequeue() + 1;

// Find a row with maximum seats

int max\_seats\_in\_a\_row = 0;

int max\_row = -1;

for (int i = 0; i < rows; i++) {

if (seats[i] > max\_seats\_in\_a\_row) {

max\_seats\_in\_a\_row = seats[i];

max\_row = i;

}

}

// If no more empty seats break out of the loop

if (max\_row == -1) {

break;

}

profit += max\_seats\_in\_a\_row;

seats[max\_row]--;

// Print current status

cout << "Total profit: " << profit << endl;

cout << "Vacant seats: ";

for (int i = 0; i < rows; i++) {

cout << seats[i] << " ";

}

cout << endl;

cout << "People in queue: " << num\_people - current << endl << endl;

}

return profit;

}

int main() {

int size;

cout << "Enter the size: ";

cin >> size;

int\* seats = new int[size];

for (int i = 0; i < size; i++) {

cout << "Enter the element of the " << i + 1 << " queue: ";

cin >> seats[i];

}

int rows, num\_people;

cout << "Enter the number of rows: ";

cin >> rows;

cout << "Enter the number of people: ";

cin >> num\_people;

cout << "Maximum profit: " << maximizeProfit(seats, num\_people, rows);

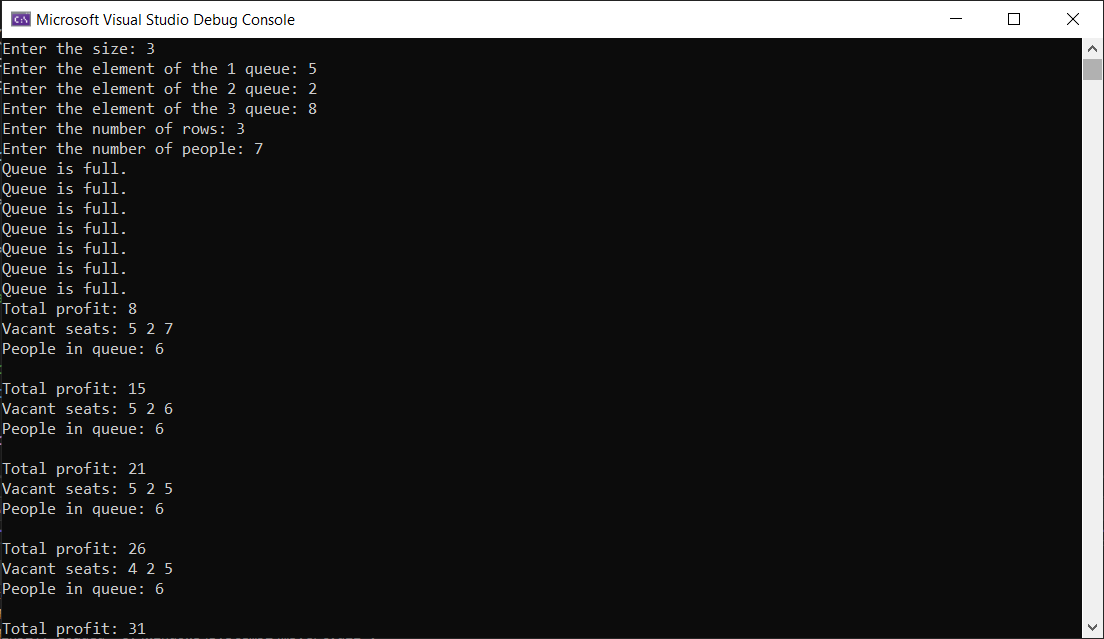
// Clean up dynamically allocated memory

delete[] seats;

return 0;

}

**Screen Shot:**

****