# RUSTAMJI INSTITUTE OF TECHNOLOGY

**BSF ACADEMY, TEKANPUR**

**Practical File for CS303 (Data Structure)**

**Submitted by**

Bhoomi sharma (0902CS231035)

B.Tech. Computer Science & Engineering 3rd Semester (2023-2027 batch)

**Subject Teacher**

Dr. Jagdish Makhijani

**File Checked by**

Mr. Yashwant Pathak



# Self-Declaration Certificate

I, **Bhoomi sharma**, hereby declare that I have completed the lab work of CS303 (Data Structure) at my own effort and understanding.

I affirm that the work submitted is my own, and I take full responsibility for its authenticity and originality.

Date:14/12/2024 [Bhoomi sharma]

[0902CS231035]

# ENVIRONMENT USED

**Hardware**

**Configuration :** < Intel Core i5 Integrated Intel Iris Xe

Graphics**>**

**C Compiler :** GCC Compiler

**User Interface :** vs code

# GROUP MEMBERS

**Member-1 :** Bhoomi sharma & 0902CS231035

https://github.com/Bhoomi1204045

**Member-2 :** Aradhana singh&0902CS231020

https://github.com/Aradhana1234

**Member-3 :** kritika singh & 0902cs231052

https://github.com/krittikaSingh123

**Member-4 :** payal chauhan & 0902CS231068

https://github.com/payalchauhan6768

# TABLE OF CONTENTS

**Section-A (Linked List)**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.**  **No.** | **Practical Description** | **Page Nos.** | **COs** |
| 1 | Implementation of Linked List using array. | 7-9 | CO-1 |
| 2 | Implementation of Linked List using Pointers. | 10-12 | CO-1 |
| 3 | Implementation of Doubly Linked List using Pointers. | 13-16 | CO-1 |
| 4 | Implementation of Circular Single Linked List using Pointers. | 17-19 | CO-1 |
| 5 | Implementation of Circular Doubly Linked List using Pointers. | 20-21 | CO-1 |

**Section-B (Stack)**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.**  **No.** | **Practical Description** | **Page Nos.** | **COs** |
| 1 | Implementation of Stack using Array. | 22-25 | CO-2 |
| 2 | Implementation of Stack using Pointers. | 26-27 | CO-2 |
| 3 | Program for Tower of Hanoi using recursion. | 28-32 | CO-2 |
| 4 | Program to find out factorial of given number using recursion. Also show the various states of stack using in this program. | 33-35 | CO-2 |

**Section-C (Queue)**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.**  **No.** | **Practical Description** | **Page Nos.** | **COs** |
| 1 | Implementation of Queue using Array. | 36-38 | CO-2 |
| 2 | Implementation of Queue using Pointers. | 39-40 | CO-2 |
| 3 | Implementation of Circular Queue using Array. | 41-42 | CO-2 |

**Section-D (Trees & Graphs)**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.**  **No.** | **Practical Description** | **Page Nos.** | **COs** |
| 1 | Implementation of Binary Search Tree. | 43-44 | CO-3 |
| 2 | Conversion of BST PreOrder/PostOrder/InOrder. | 45-46 | CO-3 |
| 3 | Implementation of Kruskal Algorithm | 47-48 | CO-4 |
| 4 | Implementation of Prim Algorithm | 49 | CO-4 |
| 5 | Implementation of Dijkstra Algorithm | 50 | CO-4 |

**Section-E (Sorting & Searching)**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.**  **No.** | **Practical Description** | **Page Nos.** | **COs** |
| 1 | Implementation of Sorting   1. Bubble 2. Selection 3. Insertion 4. Quick 5. Merge | 51-59 | CO-5 |
| 2 | Implementation of Binary Search on a list of numbers stored in an Array | 60-61 | CO-5 |
| 3 | Implementation of Binary Search on a list of strings stored in an Array | 62-63 | CO-5 |
| 4 | Implementation of Linear Search on a list of strings stored in an Array  OR  Implementation of Binary Search on a list of strings stored in a Single Linked List | 64-65 | CO-5 |

**Section-A (Linked List)**

**Experiment No.: 1**

**1.Implementation of Linked List using array.**

**Program Description:**

This program demonstrates the implementation of a linked list using an array. A traditional linked list uses dynamic memory allocation for nodes, but here we simulate the behavior of a linked list using a fixed-size array. Below is a detailed description of each component:

**Solution:**

#include <stdio.h>

#define SIZE 100 // Define the maximum size of the array

// Structure to represent a node in the array-based linked list

typedef struct {

int data; // Value stored in the node

int next; // Index of the next node

} Node;

Node list[SIZE]; // Array to hold nodes

int head = -1; // Index of the first node in the list

int freeIndex = 0; // Points to the next free position in the array

// Function to initialize the list

void initializeList() {

for (int i = 0; i < SIZE - 1; i++) {

list[i].next = i + 1;

}

list[SIZE - 1].next = -1;

}

// Function to create a new node and return its index

int createNode(int value) {

if (freeIndex == -1) {

printf("List is full!\n");

return -1;

}

int newIndex = freeIndex;

freeIndex = list[freeIndex].next; // Update free index

list[newIndex].data = value;

list[newIndex].next = -1;

return newIndex;

}

// Function to insert a value at the beginning

void insertAtBeginning(int value) {

int newNode = createNode(value);

if (newNode != -1) {

list[newNode].next = head;

head = newNode;

}

}

// Function to delete a node with a specific value

void deleteNode(int value) {

if (head == -1) {

printf("List is empty!\n");

return;

}

int current = head, prev = -1;

while (current != -1 && list[current].data != value) {

prev = current;

current = list[current].next;

}

if (current == -1) {

printf("Value not found in the list!\n");

return;

}

if (prev == -1) {

head = list[current].next;

} else {

list[prev].next = list[current].next;

}

list[current].next = freeIndex; // Add the node back to the free list

freeIndex = current;

}

// Function to display the linked list

void displayList() {

if (head == -1) {

printf("List is empty!\n");

return;

}

int current = head;

printf("List: ");

while (current != -1) {

printf("%d -> ", list[current].data);

current = list[current].next;

}

printf("NULL\n");

}

// Main function

int main() {

initializeList();

insertAtBeginning(10);

insertAtBeginning(20);

insertAtBeginning(30);

printf("After inserting 30, 20, 10 at the beginning:\n");

displayList();

deleteNode(20);

printf("After deleting 20:\n");

displayList();

deleteNode(50);

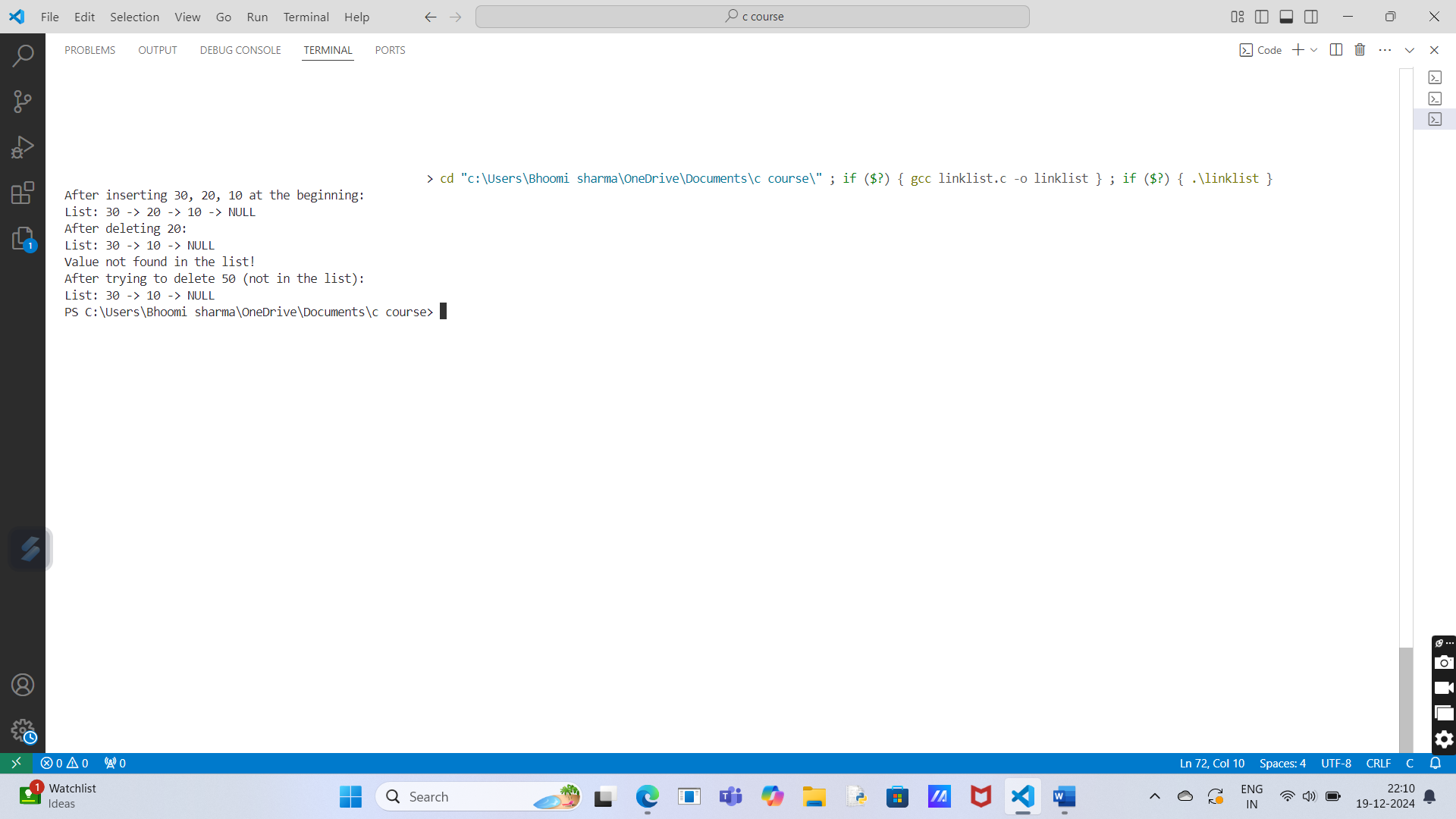
printf("After trying to delete 50 (not in the list):\n");

displayList();

return 0;

}

**Output:**



**2.Implementation of Linked List using Pointers.**

**Program Description:**

This C program implements a Singly Linked List using pointers. A singly linked list is a data structure in which each element (node) contains two parts:

1. Data: The actual value or information that the node holds.
2. Next: A pointer that refers to the next node in the list (or NULL if it’s the last node).

**Solution:**

#include <stdio.h>

#include <stdlib.h>

// Node structure

struct Node {

int data;

struct Node\* next;

};

// Function to create a new node

struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

newNode->next = NULL;

return newNode;

}

// Function to add a node at the end

void appendNode(struct Node\*\* head, int data) {

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

return;

}

struct Node\* temp = \*head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

// Function to display the linked list

void displayList(struct Node\* head) {

if (head == NULL) {

printf("The list is empty.\n");

return;

}

struct Node\* temp = head;

printf("Linked List: ");

while (temp != NULL) {

printf("%d -> ", temp->data);

temp = temp->next;

}

printf("NULL\n");

}

// Function to delete a node by value

void deleteNode(struct Node\*\* head, int value) {

struct Node\* temp = \*head;

struct Node\* prev = NULL;

if (temp != NULL && temp->data == value) {

\*head = temp->next;

free(temp);

printf("Node with value %d deleted.\n", value);

return;

}

while (temp != NULL && temp->data != value) {

prev = temp;

temp = temp->next;

}

if (temp == NULL) {

printf("Value %d not found in the list.\n", value);

return;

}

prev->next = temp->next;

free(temp);

printf("Node with value %d deleted.\n", value);

}

// Main function

int main() {

struct Node\* head = NULL;

// Adding nodes

appendNode(&head, 10);

appendNode(&head, 20);

appendNode(&head, 30);

// Display the list

displayList(head);

// Delete a node

deleteNode(&head, 20);

displayList(head);

// Delete a non-existing node

deleteNode(&head, 40);

return 0;

}

**Output:**



**3.Implementation of Doubly Linked List using Pointers.**

**Program Description:**

This C program demonstrates the implementation of a Doubly Linked List (DLL). In a DLL, each node contains three parts:

1. Data: Holds the actual value stored in the node.
2. Next: Pointer to the next node.
3. Prev: Pointer to the previous node.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

// Node structure

struct Node {

int data;

struct Node\* next;

struct Node\* prev;

};

// Function to create a new node

struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

newNode->next = NULL;

newNode->prev = NULL;

return newNode;

}

// Function to add a node at the end of the doubly linked list

void appendNode(struct Node\*\* head, int data) {

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

return;

}

struct Node\* temp = \*head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

// Function to display the doubly linked list in forward direction

void displayListForward(struct Node\* head) {

if (head == NULL) {

printf("The list is empty.\n");

return;

}

struct Node\* temp = head;

printf("Doubly Linked List (Forward): ");

while (temp != NULL) {

printf("%d <-> ", temp->data);

temp = temp->next;

}

printf("NULL\n");

}

// Function to display the doubly linked list in reverse direction

void displayListBackward(struct Node\* head) {

if (head == NULL) {

printf("The list is empty.\n");

return;

}

struct Node\* temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

printf("Doubly Linked List (Backward): ");

while (temp != NULL) {

printf("%d <-> ", temp->data);

temp = temp->prev;

}

printf("NULL\n");

}

// Function to delete a node by value

void deleteNode(struct Node\*\* head, int value) {

struct Node\* temp = \*head;

// If the list is empty

if (\*head == NULL) {

printf("The list is empty.\n");

return;

}

// If the node to be deleted is the head node

if (temp != NULL && temp->data == value) {

\*head = temp->next;

if (\*head != NULL) {

(\*head)->prev = NULL;

}

free(temp);

printf("Node with value %d deleted.\n", value);

return;

}

// Search for the node to delete

while (temp != NULL && temp->data != value) {

temp = temp->next;

}

// If the node is not found

if (temp == NULL) {

printf("Node with value %d not found.\n", value);

return;

}

// Unlink the node from the doubly linked list

if (temp->next != NULL) {

temp->next->prev = temp->prev;

}

if (temp->prev != NULL) {

temp->prev->next = temp->next;

}

free(temp);

printf("Node with value %d deleted.\n", value);

}

// Main function

int main() {

struct Node\* head = NULL;

// Adding nodes

appendNode(&head, 10);

appendNode(&head, 20);

appendNode(&head, 30);

// Display the list forward

displayListForward(head);

// Display the list backward

displayListBackward(head);

// Delete a node

deleteNode(&head, 20);

// Display the list after deletion

displayListForward(head);

displayListBackward(head);

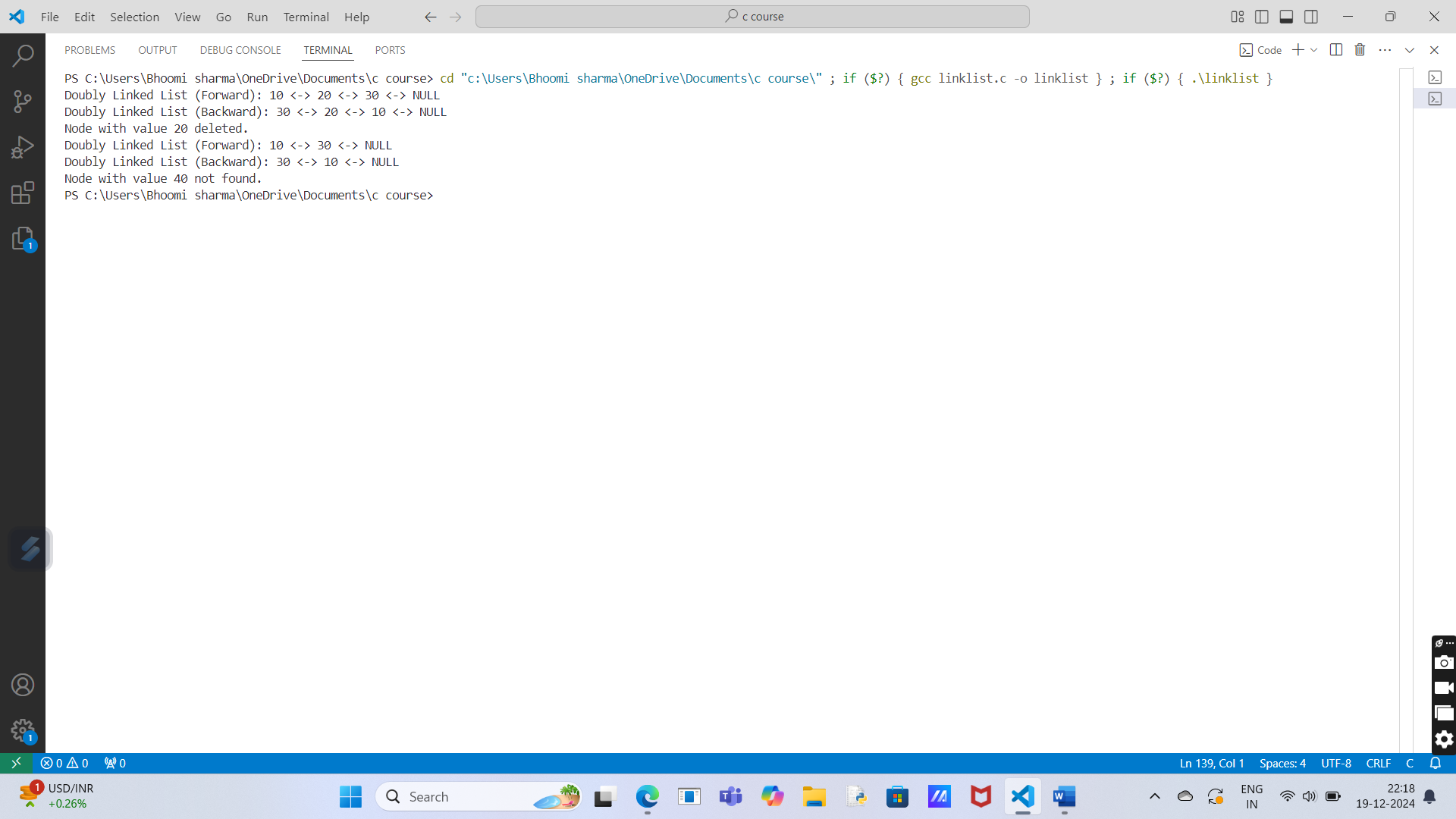
// Delete a non-existing node

deleteNode(&head, 40);

return 0;

}

**Output:**

****

**4.Implementation of Circular Single Linked List using Pointers.**

**Program Description:**

This C program implements a Circular Singly Linked List using pointers. It defines a Node structure with an integer data field and a pointer to the next node. The program includes functions to:

1. Create a new node with a given value, linking it to itself.
2. Insert a node at the end of the circular linked list, adjusting pointers to maintain the circular structure.
3. Display the list by traversing it starting from the head node until it loops back to the head.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

// Define the structure of a node

struct Node {

int data;

struct Node\* next;

};

// Function to create a new node

struct Node\* createNode(int value) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

newNode->next = newNode; // Circular link (points to itself)

return newNode;

}

// Function to insert a new node at the end of the circular linked list

void insertEnd(struct Node\*\* head, int value) {

struct Node\* newNode = createNode(value);

if (\*head == NULL) {

\*head = newNode; // If the list is empty, the new node becomes the head

} else {

struct Node\* temp = \*head;

while (temp->next != \*head) { // Traverse until we reach the last node

temp = temp->next;

}

temp->next = newNode; // Last node's next points to new node

newNode->next = \*head; // New node points to the head

}

}

// Function to display the circular linked list

void display(struct Node\* head) {

if (head == NULL) {

printf("List is empty.\n");

return;

}

struct Node\* temp = head;

do {

printf("%d -> ", temp->data);

temp = temp->next;

} while (temp != head); // Loop until we come back to the head

printf("(head)\n");

}

// Function to delete the entire circular linked list

void deleteList(struct Node\*\* head) {

if (\*head == NULL) {

return;

}

struct Node\* temp = \*head;

struct Node\* nextNode = NULL;

do {

nextNode = temp->next;

free(temp);

temp = nextNode;

} while (temp != \*head);

\*head = NULL; // Set the head to NULL as the list is deleted

}

// Main function to test the Circular Linked List

int main() {

struct Node\* head = NULL;

// Insert nodes at the end of the circular linked list

insertEnd(&head, 10);

insertEnd(&head, 20);

insertEnd(&head, 30);

insertEnd(&head, 40);

// Display the circular linked list

printf("Circular Linked List: \n");

display(head);

// Deleting the list

deleteList(&head);

printf("List after deletion: \n");

display(head);

return 0;

}

**Output:**



**5.Implementation of Circular Doubly Linked List using Pointers.**

**Program Description:**

This program creates a circular doubly linked list that allows for insertion of nodes at the end and traversal in both directions (forward and backward). It uses a structure Node to store the data and pointers to the next and previous nodes. The head pointer represents the start of the list.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

// Define the structure for the node

struct Node {

int data;

struct Node\* next;

struct Node\* prev;

};

// Function to create a new node

struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

newNode->next = newNode->prev = newNode; // Pointing to itself initially

return newNode;

}

// Function to insert a node at the end

void insertEnd(struct Node\*\* head, int data) {

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

} else {

struct Node\* temp = \*head;

while (temp->next != \*head) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

newNode->next = \*head;

(\*head)->prev = newNode;

}

}

// Function to print the list in forward direction

void printListForward(struct Node\* head) {

if (head == NULL) {

printf("List is empty.\n");

return;

}

struct Node\* temp = head;

do {

printf("%d ", temp->data);

temp = temp->next;

} while (temp != head);

printf("\n");

}

// Function to print the list in backward direction

void printListBackward(struct Node\* head) {

if (head == NULL) {

printf("List is empty.\n");

return;

}

struct Node\* temp = head->prev;

do {

printf("%d ", temp->data);

temp = temp->prev;

} while (temp != head->prev);

printf("\n");

}

// Main function

int main() {

struct Node\* head = NULL;

// Inserting nodes at the end

insertEnd(&head, 10);

insertEnd(&head, 20);

insertEnd(&head, 30);

insertEnd(&head, 40);

// Printing the list in forward direction

printf("List in forward direction: ");

printListForward(head);

// Printing the list in backward direction

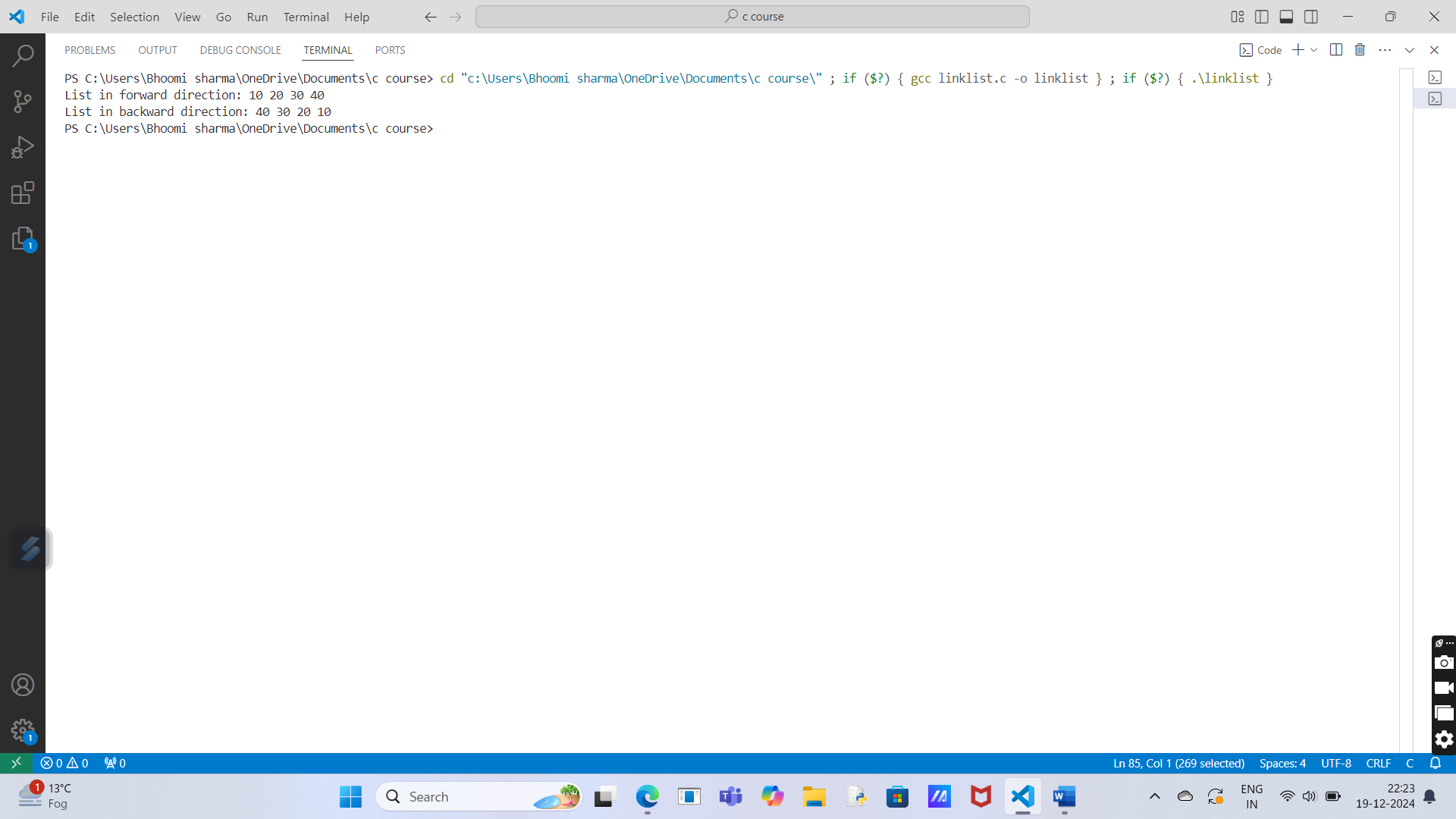
printf("List in backward direction: ");

printListBackward(head);

return 0;

}

**Output:**



**Section-B (Stack)**

**Experiment No.: 2**

**1.Implementation of stack using array.**

**Program Description:**

A stack is a linear data structure that follows the Last In First Out (LIFO) principle. It supports two primary operations:

1. Push: Add an element to the stack.
2. Pop: Remove the top element from the stack.

We implement the stack using a fixed-size array, and we manage the top of the stack using an integer.

**Solution:**

#include <stdio.h>

#define MAX 5 // Define maximum size of the stack

int stack[MAX];

int top = -1; // Initialize top of stack to -1 (empty stack)

void push(int value) {

if (top == MAX - 1) {

printf("Stack Overflow\n"); // Stack is full

} else {

stack[++top] = value; // Increment top and push value

printf("%d pushed to stack\n", value);

}

}

int pop() {

if (top == -1) {

printf("Stack Underflow\n"); // Stack is empty

return -1;

} else {

printf("%d popped from stack\n", stack[top]);

return stack[top--]; // Return top value and decrement top

}

}

void display() {

if (top == -1) {

printf("Stack is empty\n");

} else {

printf("Stack elements: ");

for (int i = top; i >= 0; i--) {

printf("%d ", stack[i]);

}

printf("\n");

}

}

int main() {

push(10); // Push elements onto the stack

push(20);

push(30);

display(); // Display stack

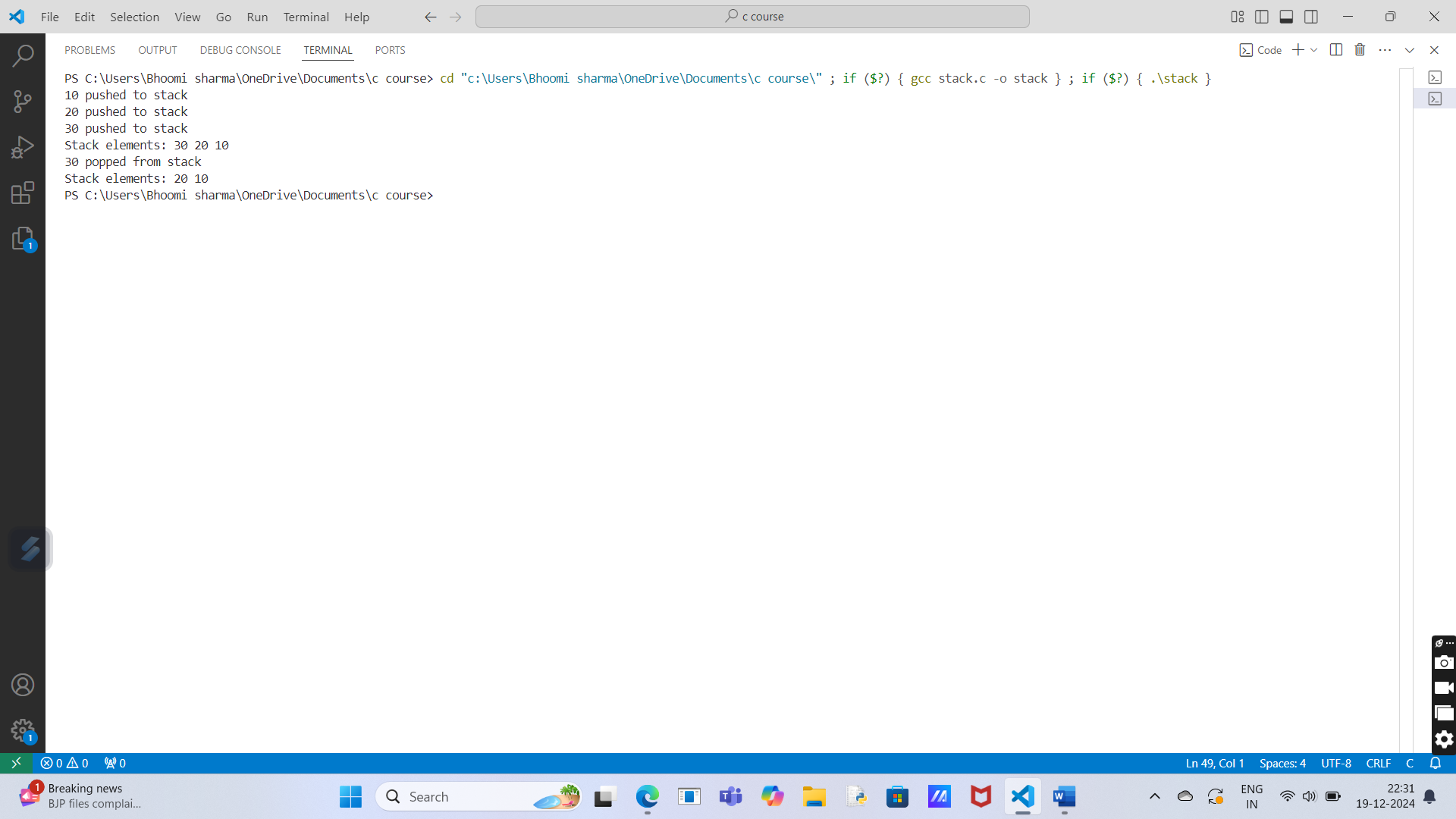
pop(); // Pop an element

display(); // Display stack after pop

return 0;

}

**Output:**



**2.Implementation of Stack using Pointers.**

**Program Description:**

we will implement a stack using pointers. A stack is a linear data structure that follows the Last In, First Out (LIFO) principle, where elements are added or removed from one end, called the "top." The program uses dynamic memory allocation for the stack and performs the standard stack operations: push and pop.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

// Define the structure for the stack

struct Stack {

int data;

struct Stack \*next;

};

// Function to push an element onto the stack

void push(struct Stack \*\*top, int value) {

struct Stack \*newNode = (struct Stack\*)malloc(sizeof(struct Stack));

if (!newNode) {

printf("Memory allocation failed!\n");

return;

}

newNode->data = value;

newNode->next = \*top;

\*top = newNode;

printf("Pushed %d onto the stack.\n", value);

}

// Function to pop an element from the stack

int pop(struct Stack \*\*top) {

if (\*top == NULL) {

printf("Stack underflow! Stack is empty.\n");

return -1;

}

struct Stack \*temp = \*top;

int poppedValue = temp->data;

\*top = (\*top)->next;

free(temp);

return poppedValue;

}

// Function to display the top element of the stack

int peek(struct Stack \*top) {

if (top == NULL) {

printf("Stack is empty.\n");

return -1;

}

return top->data;

}

// Main function to test the stack operations

int main() {

struct Stack \*top = NULL;

// Push elements onto the stack

push(&top, 10);

push(&top, 20);

push(&top, 30);

// Display the top element of the stack

printf("Top element is: %d\n", peek(top));

// Pop elements from the stack

printf("Popped element: %d\n", pop(&top));

printf("Popped element: %d\n", pop(&top));

printf("Popped element: %d\n", pop(&top));

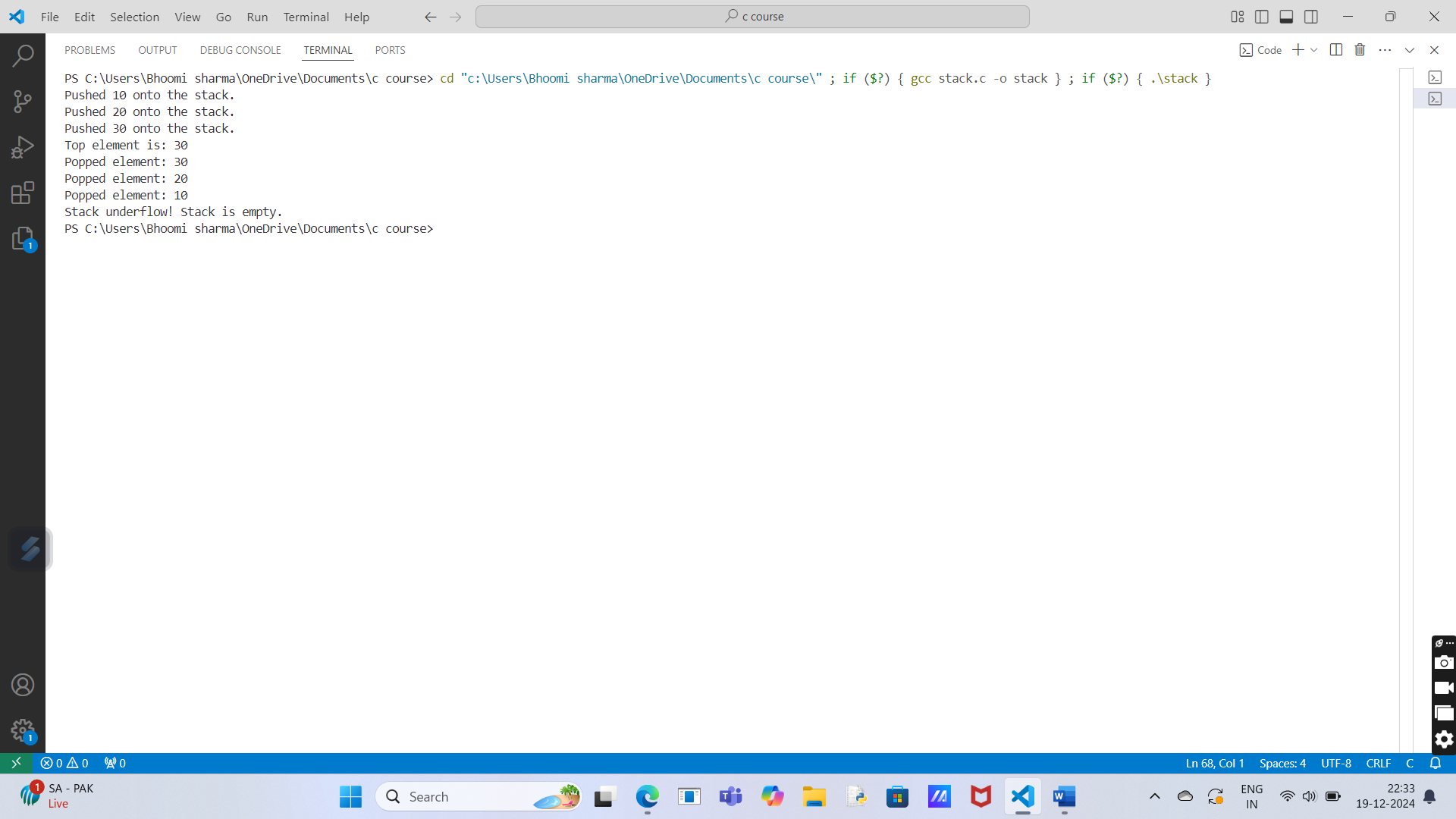
// Attempt to pop from an empty stack

pop(&top);

return 0;

}

**Output:**



**3.Program for Tower of Hanoi using recursion.**

**Program Description:**

This program solves the Tower of Hanoi problem using recursion. The objective is to move n disks from a source peg to a destination peg, following the rules that only one disk can be moved at a time and a larger disk cannot be placed on top of a smaller one. The program recursively breaks down the problem into smaller sub-problems, moving n-1 disks to an auxiliary peg before moving the nth disk to the destination, and then moving the n-1 disks onto the destination peg. The sequence of moves is printed step by step

**Solution:**

#include <stdio.h>

// Function to move 'n' disks from source to destination using auxiliary peg

void towerOfHanoi(int n, char source, char destination, char auxiliary) {

// Base case: If only one disk is left

if (n == 1) {

printf("Move disk 1 from %c to %c\n", source, destination);

return;

}

// Move 'n-1' disks from source to auxiliary peg

towerOfHanoi(n - 1, source, auxiliary, destination);

// Move the nth disk from source to destination peg

printf("Move disk %d from %c to %c\n", n, source, destination);

// Move the 'n-1' disks from auxiliary to destination peg

towerOfHanoi(n - 1, auxiliary, destination, source);

}

// Main function to test the Tower of Hanoi solution

int main() {

int n;

// Ask the user for the number of disks

printf("Enter the number of disks: ");

scanf("%d", &n);

// Call the recursive function to solve the Tower of Hanoi problem

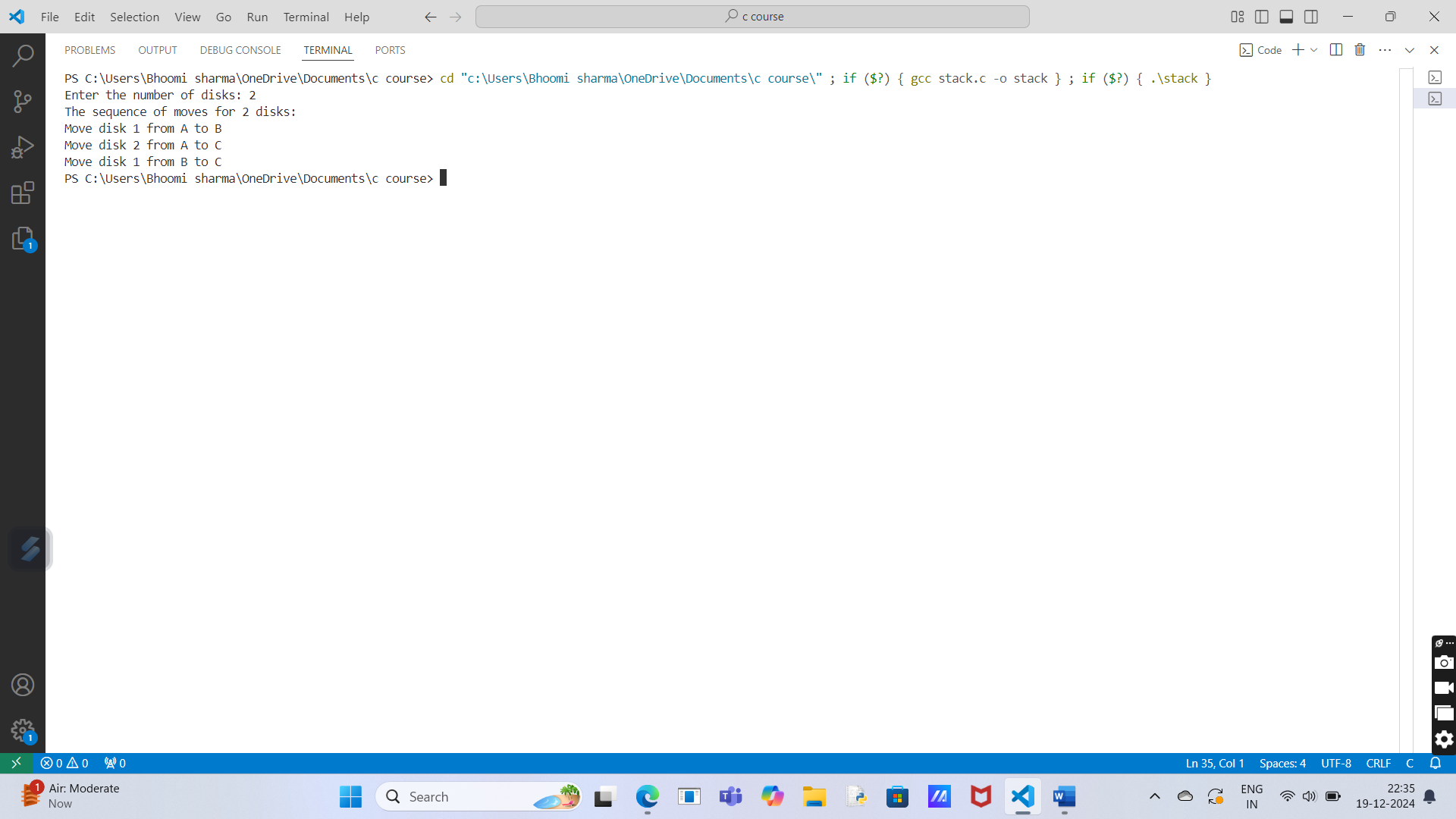
printf("The sequence of moves for %d disks:\n", n);

towerOfHanoi(n, 'A', 'C', 'B'); // A = source, C = destination, B = auxiliary

return 0;

}

**Output:**



**4.Program to find out factorial of given number using recursion. Also show the various states of stack using in this program.**

**Program Description:**

This program calculates the factorial of a given number n using recursion. The factorial of a number n, denoted by n!, is the product of all positive integers less than or equal to n. The factorial is calculated recursively as:

* n! = n \* (n-1)! for n > 1
* 1! = 1 (base case)

The program also displays the states of the call stack at each recursive call to help visualize the recursion process.

**Solution:**

#include <stdio.h>

// Function to calculate factorial recursively

int factorial(int n) {

printf("Calling factorial(%d)\n", n); // Print current state of stack

if (n == 0 || n == 1) {

return 1; // Base case: factorial(0) = 1

}

return n \* factorial(n - 1); // Recursive call

}

// Main function to test the factorial program

int main() {

int num;

// Ask the user for input

printf("Enter a number: ");

scanf("%d", &num);

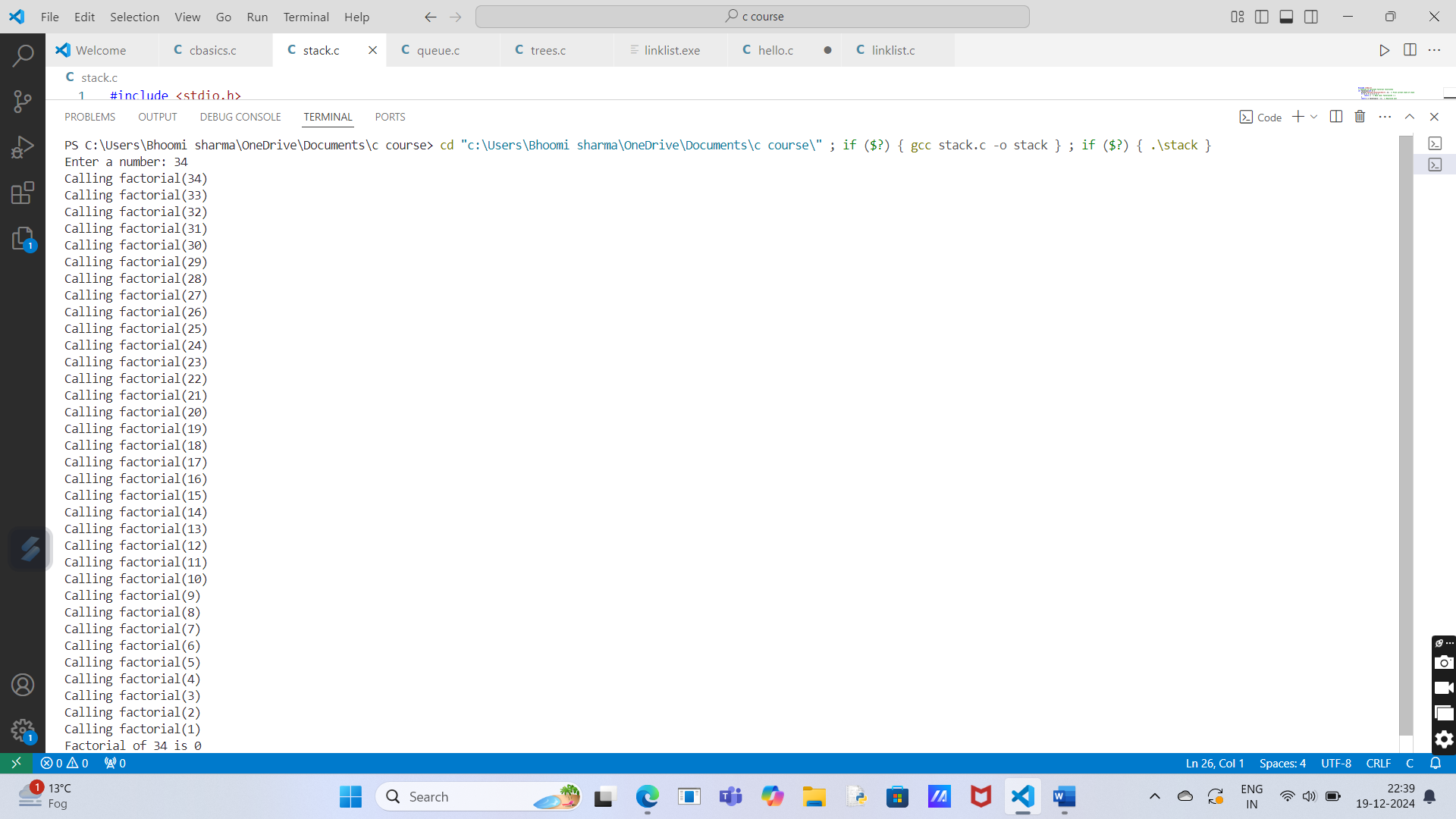
// Calculate factorial and display result

printf("Factorial of %d is %d\n", num, factorial(num));

return 0;

}

**Output:**



**Section-C (Queue)**

**Experiment No.: 3**

**1.Implementation of Queue using Array.**

**Program Description:**

This program implements a queue using an array. A queue is a linear data structure that follows the First In, First Out (FIFO) principle. In this implementation, the program supports the basic operations of a queue:

* **Enqueue**: Adds an element to the rear of the queue.
* **Dequeue**: Removes an element from the front of the queue.
* **Display**: Displays all elements in the queue.
* **Front**: Displays the front element of the queue.

The queue uses an array to store the elements, and we maintain two pointers:

* front: Points to the front element of the queue.
* rear: Points to the last element of the queue.

**Solution:**

#include <stdio.h>

#define MAX 5 // Define the maximum size of the queue

// Structure to represent a queue

struct Queue {

int arr[MAX];

int front, rear;

};

// Function to initialize the queue

void initializeQueue(struct Queue \*q) {

q->front = -1;

q->rear = -1;

}

// Function to check if the queue is full

int isFull(struct Queue \*q) {

return (q->rear == MAX - 1);

}

// Function to check if the queue is empty

int isEmpty(struct Queue \*q) {

return (q->front == -1 || q->front > q->rear);

}

// Function to add an element to the queue

void enqueue(struct Queue \*q, int value) {

if (isFull(q)) {

printf("Queue is full! Cannot enqueue.\n");

} else {

if (q->front == -1) {

q->front = 0; // First element added

}

q->rear++;

q->arr[q->rear] = value;

printf("Enqueued %d\n", value);

}

}

// Function to remove an element from the queue

int dequeue(struct Queue \*q) {

if (isEmpty(q)) {

printf("Queue is empty! Cannot dequeue.\n");

return -1; // Return -1 to indicate an error

} else {

int value = q->arr[q->front];

q->front++;

if (q->front > q->rear) {

q->front = q->rear = -1; // Reset the queue if it becomes empty

}

return value;

}

}

// Function to display the elements of the queue

void display(struct Queue \*q) {

if (isEmpty(q)) {

printf("Queue is empty!\n");

} else {

printf("Queue elements: ");

for (int i = q->front; i <= q->rear; i++) {

printf("%d ", q->arr[i]);

}

printf("\n");

}

}

// Main function to test the queue implementation

int main() {

struct Queue q;

initializeQueue(&q);

// Enqueue elements

enqueue(&q, 10);

enqueue(&q, 20);

enqueue(&q, 30);

enqueue(&q, 40);

enqueue(&q, 50);

// Display queue

display(&q);

// Enqueue when the queue is full

enqueue(&q, 60);

// Dequeue elements

printf("Dequeued: %d\n", dequeue(&q));

printf("Dequeued: %d\n", dequeue(&q));

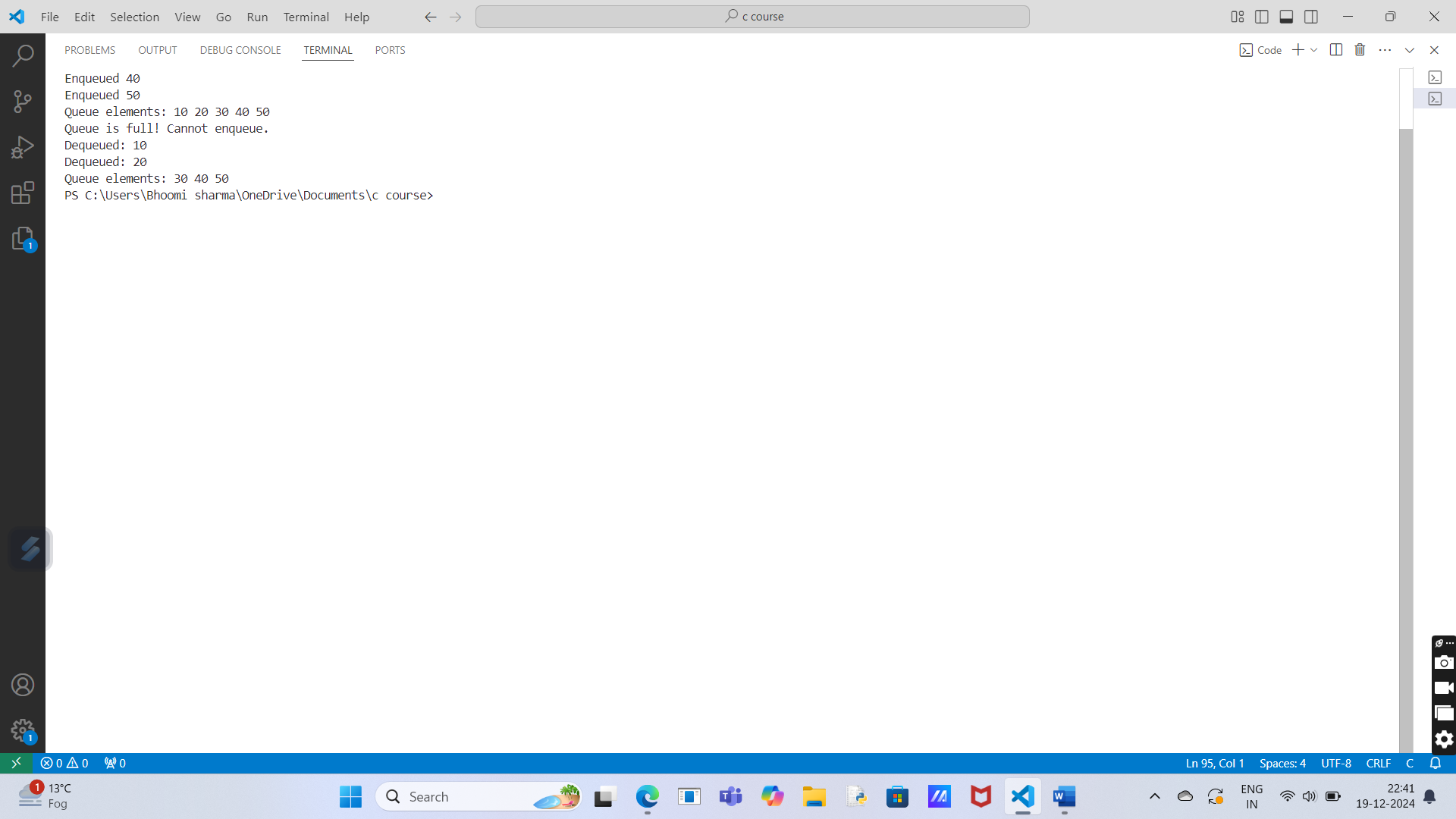
// Display queue after dequeue

display(&q);

return 0;

}

**Output:**



**2.Implementation of Queue using Pointers.**

**Program Description:**

This program implements a queue using pointers, which allows dynamic memory allocation and resizing of the queue. A queue follows the First In, First Out (FIFO) principle. The program uses a linked list structure, where each node in the list represents an element in the queue.

In this implementation, we have the following operations:

* Enqueue: Adds an element to the rear of the queue.
* Dequeue: Removes an element from the front of the queue.
* Display: Displays all elements in the queue.

The program defines a Node structure that contains the data and a pointer to the next node. The queue uses two pointers, front and rear, to keep track of the front and rear elements**.**

**Solution:**

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a queue node

struct Node {

    int data;

    struct Node \*next;

};

// Define the structure for the queue

struct Queue {

    struct Node \*front, \*rear;

};

// Function to initialize the queue

void initializeQueue(struct Queue \*q) {

    q->front = q->rear = NULL;

}

// Function to check if the queue is empty

int isEmpty(struct Queue \*q) {

    return q->front == NULL;

}

// Function to add an element to the queue

void enqueue(struct Queue \*q, int value) {

    struct Node \*newNode = (struct Node\*)malloc(sizeof(struct Node));

    if (newNode == NULL) {

        printf("Memory allocation failed!\n");

        return;

    }

    newNode->data = value;

    newNode->next = NULL;

    if (isEmpty(q)) {

        q->front = q->rear = newNode;

    } else {

        q->rear->next = newNode;

        q->rear = newNode;

    }

    printf("Enqueued %d\n", value);

}

// Function to remove an element from the queue

int dequeue(struct Queue \*q) {

    if (isEmpty(q)) {

        printf("Queue is empty! Cannot dequeue.\n");

        return -1;

    }

    struct Node \*temp = q->front;

    int value = temp->data;

    q->front = q->front->next;

    if (q->front == NULL) {

        q->rear = NULL; // If the queue is empty, set rear to NULL

    }

    free(temp);

    return value;

}

// Function to display the elements of the queue

void display(struct Queue \*q) {

    if (isEmpty(q)) {

        printf("Queue is empty!\n");

        return;

    }

    struct Node \*temp = q->front;

    printf("Queue elements: ");

    while (temp != NULL) {

        printf("%d ", temp->data);

        temp = temp->next;

    }

    printf("\n");

}

// Main function to test the queue implementation

int main() {

    struct Queue q;

    initializeQueue(&q);

    // Enqueue elements

    enqueue(&q, 10);

    enqueue(&q, 20);

    enqueue(&q, 30);

    enqueue(&q, 40);

    enqueue(&q, 50);

    // Display queue

    display(&q);

    // Dequeue elements

    printf("Dequeued: %d\n", dequeue(&q));

    printf("Dequeued: %d\n", dequeue(&q));

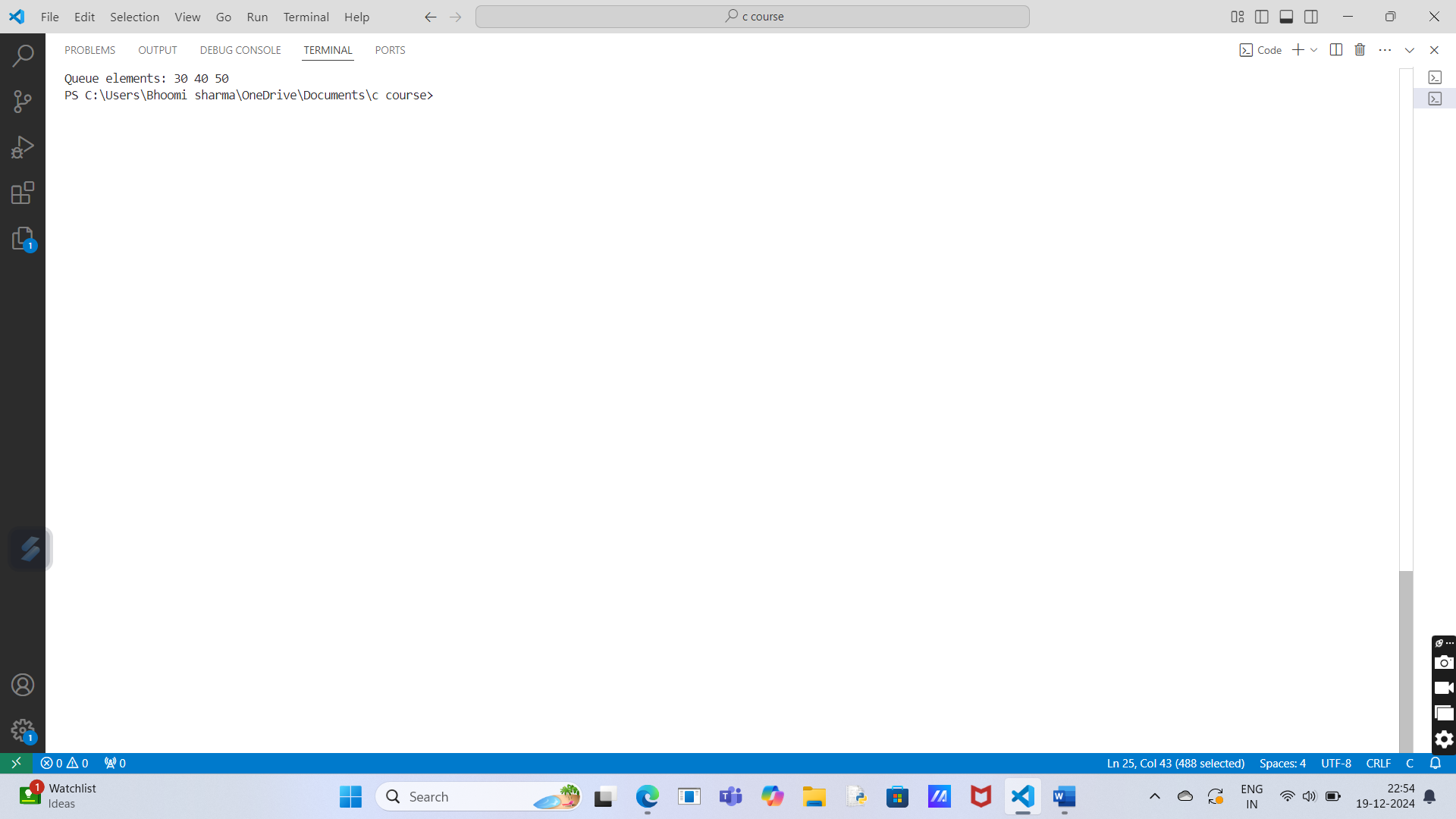
    // Display queue after dequeue

    display(&q);

    return 0;

}

**Output:**



**3.Implementation of Circular Queue using Array.**

**Program Description:**

A circular queue is a queue where the last position is connected back to the first position to form a circle. This helps in utilizing the space more efficiently in the array. In this implementation, we have a fixed-size array that acts as the storage for the queue, and the front and rear pointers are used to track the positions in the queue.

**Solution:**

#include <stdio.h>

#define MAX 5 // Define the maximum size of the circular queue

// Structure to represent a circular queue

struct Queue {

int arr[MAX];

int front, rear;

};

// Function to initialize the queue

void initializeQueue(struct Queue \*q) {

q->front = q->rear = -1;

}

// Function to check if the queue is full

int isFull(struct Queue \*q) {

return ((q->rear + 1) % MAX == q->front);

}

// Function to check if the queue is empty

int isEmpty(struct Queue \*q) {

return (q->front == -1);

}

// Function to add an element to the queue

void enqueue(struct Queue \*q, int value) {

if (isFull(q)) {

printf("Queue is full! Cannot enqueue.\n");

} else {

if (q->front == -1) { // Queue is empty, set front to 0

q->front = 0;

}

q->rear = (q->rear + 1) % MAX; // Circular increment

q->arr[q->rear] = value;

printf("Enqueued %d\n", value);

}

}

// Function to remove an element from the queue

int dequeue(struct Queue \*q) {

if (isEmpty(q)) {

printf("Queue is empty! Cannot dequeue.\n");

return -1;

} else {

int value = q->arr[q->front];

if (q->front == q->rear) { // Only one element in the queue

q->front = q->rear = -1; // Reset the queue

} else {

q->front = (q->front + 1) % MAX; // Circular increment

}

return value;

}

}

// Function to display the elements of the queue

void display(struct Queue \*q) {

if (isEmpty(q)) {

printf("Queue is empty!\n");

} else {

printf("Queue elements: ");

int i = q->front;

while (i != q->rear) {

printf("%d ", q->arr[i]);

i = (i + 1) % MAX; // Circular increment

}

printf("%d\n", q->arr[q->rear]); // Print the rear element

}

}

// Main function to test the circular queue implementation

int main() {

struct Queue q;

initializeQueue(&q);

// Enqueue elements

enqueue(&q, 10);

enqueue(&q, 20);

enqueue(&q, 30);

enqueue(&q, 40);

enqueue(&q, 50);

// Display queue

display(&q);

// Enqueue when the queue is full

enqueue(&q, 60);

// Dequeue elements

printf("Dequeued: %d\n", dequeue(&q));

printf("Dequeued: %d\n", dequeue(&q));

// Display queue after dequeue

display(&q);

// Enqueue more elements to test the circular behavior

enqueue(&q, 60);

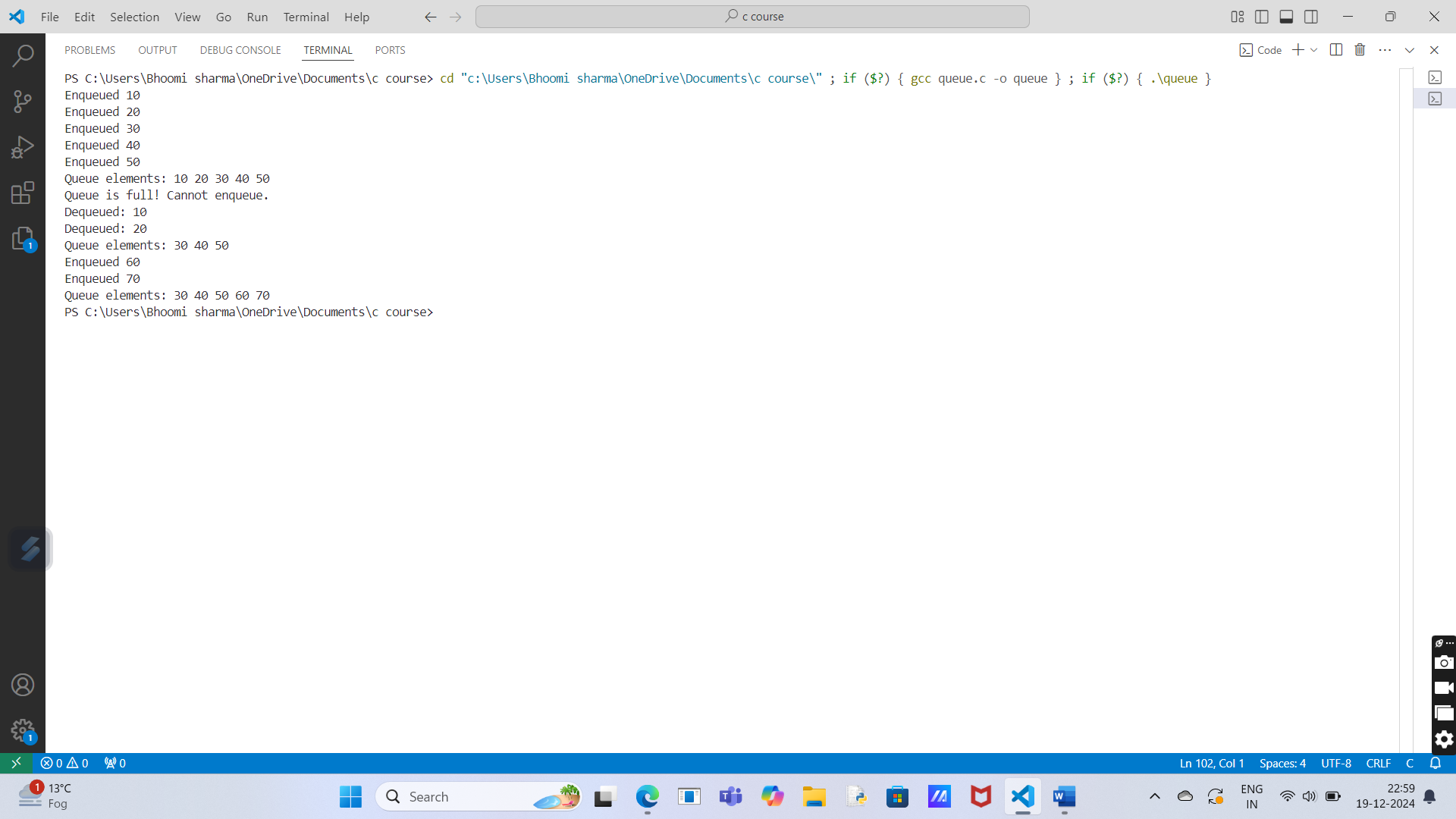
enqueue(&q, 70);

display(&q);

return 0;

}

**Output:**



**Section-D (Trees & Graphs)**

**Experiment No.: 4**

**1.Implementation of Binary Search Tree.**

**Program Description:**

This program implements a Binary Search Tree (BST) in C. It allows the insertion of integers into the tree and displays them in sorted order using inorder traversal. The program defines a Node structure with fields for data, left, and right child pointers. Key functions include:

1. createNode: Creates a new tree node.
2. insert: Adds a value to the BST while maintaining its properties.
3. inorderTraversal: Recursively prints the tree in ascending order.

The main function demonstrates the insertion of sample values and prints the sorted output.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

// Define the structure of a tree node

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

// Create a new node

struct Node\* createNode(int value) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Insert a value into the BST

struct Node\* insert(struct Node\* root, int value) {

if (root == NULL) {

return createNode(value);

}

if (value < root->data) {

root->left = insert(root->left, value);

} else if (value > root->data) {

root->right = insert(root->right, value);

}

return root;

}

// Inorder traversal (prints the tree in sorted order)

void inorderTraversal(struct Node\* root) {

if (root != NULL) {

inorderTraversal(root->left);

printf("%d ", root->data);

inorderTraversal(root->right);

}

}

// Main function

int main() {

struct Node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 70);

insert(root, 20);

insert(root, 40);

insert(root, 60);

insert(root, 80);

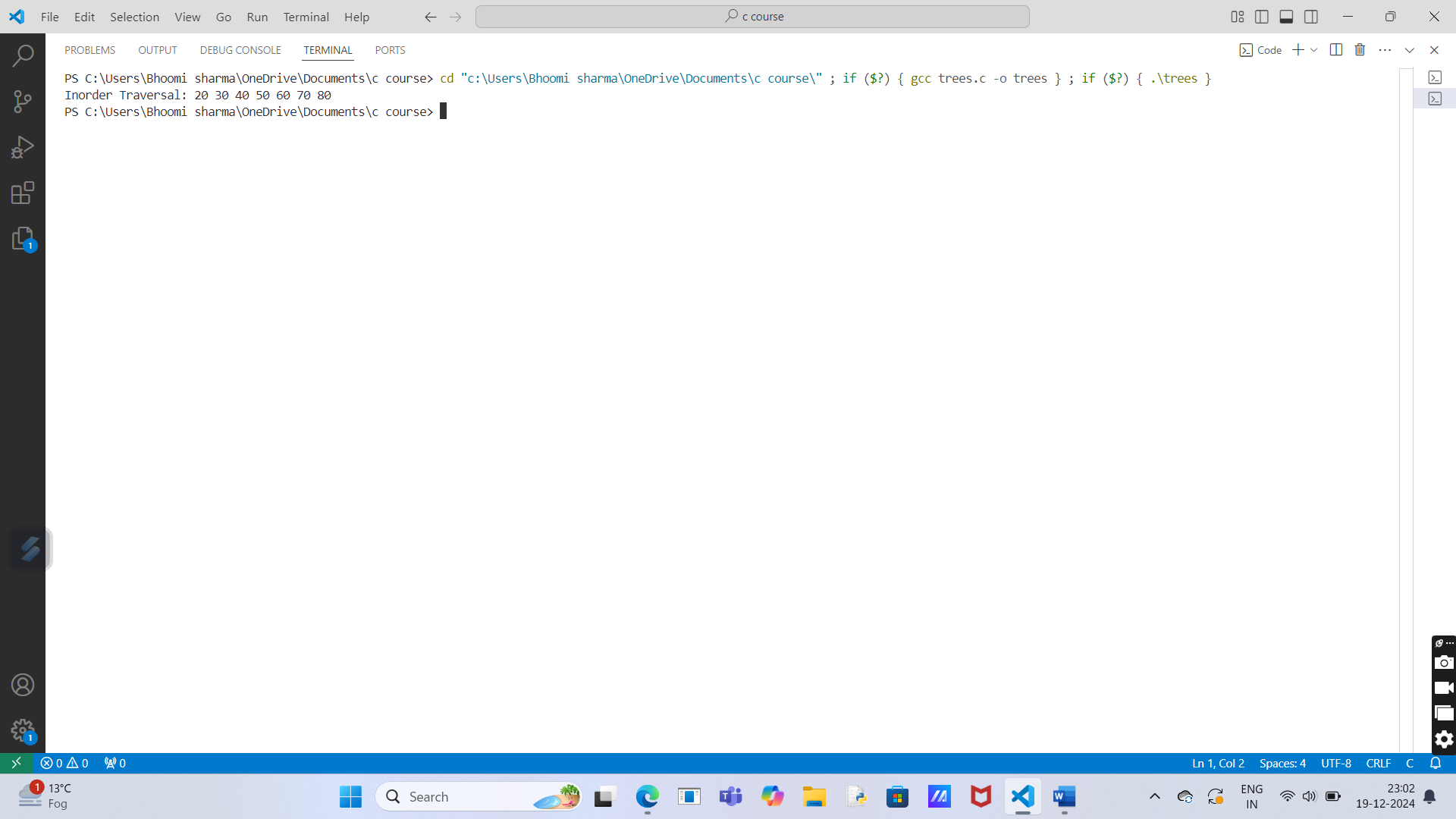
printf("Inorder Traversal: ");

inorderTraversal(root);

return 0;

}

**Output:**



**2.Conversion of BST PreOrder/PostOrder/InOrder.**

**Program Description:**

This program constructs a Binary Search Tree (BST) and performs three types of tree traversals:

1. InOrder Traversal (Left -> Root -> Right): Outputs nodes in ascending order.
2. PreOrder Traversal (Root -> Left -> Right): Useful for recreating the tree structure.
3. PostOrder Traversal (Left -> Right -> Root): Often used for deleting or processing the tree.

The program:

1. Accepts user input to create the BST.
2. Traverses the BST using the above methods.
3. Outputs the traversal results in a simple format.

It uses recursion for traversal and compact code for efficiency.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

// Node structure

struct Node {

int data;

struct Node \*left, \*right;

};

// Create a new node

struct Node\* createNode(int data) {

struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));

node->data = data;

node->left = node->right = NULL;

return node;

}

// Insert into BST

struct Node\* insert(struct Node\* root, int data) {

if (!root) return createNode(data);

if (data < root->data) root->left = insert(root->left, data);

else if (data > root->data) root->right = insert(root->right, data);

return root;

}

// Traversals

void inOrder(struct Node\* root) { if (root) { inOrder(root->left); printf("%d ", root->data); inOrder(root->right); } }

void preOrder(struct Node\* root) { if (root) { printf("%d ", root->data); preOrder(root->left); preOrder(root->right); } }

void postOrder(struct Node\* root) { if (root) { postOrder(root->left); postOrder(root->right); printf("%d ", root->data); } }

// Main function

int main() {

struct Node\* root = NULL;

int n, val;

printf("Number of elements: ");

scanf("%d", &n);

printf("Enter elements: ");

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

scanf("%d", &val);

root = insert(root, val);

}

printf("InOrder: "); inOrder(root); printf("\n");

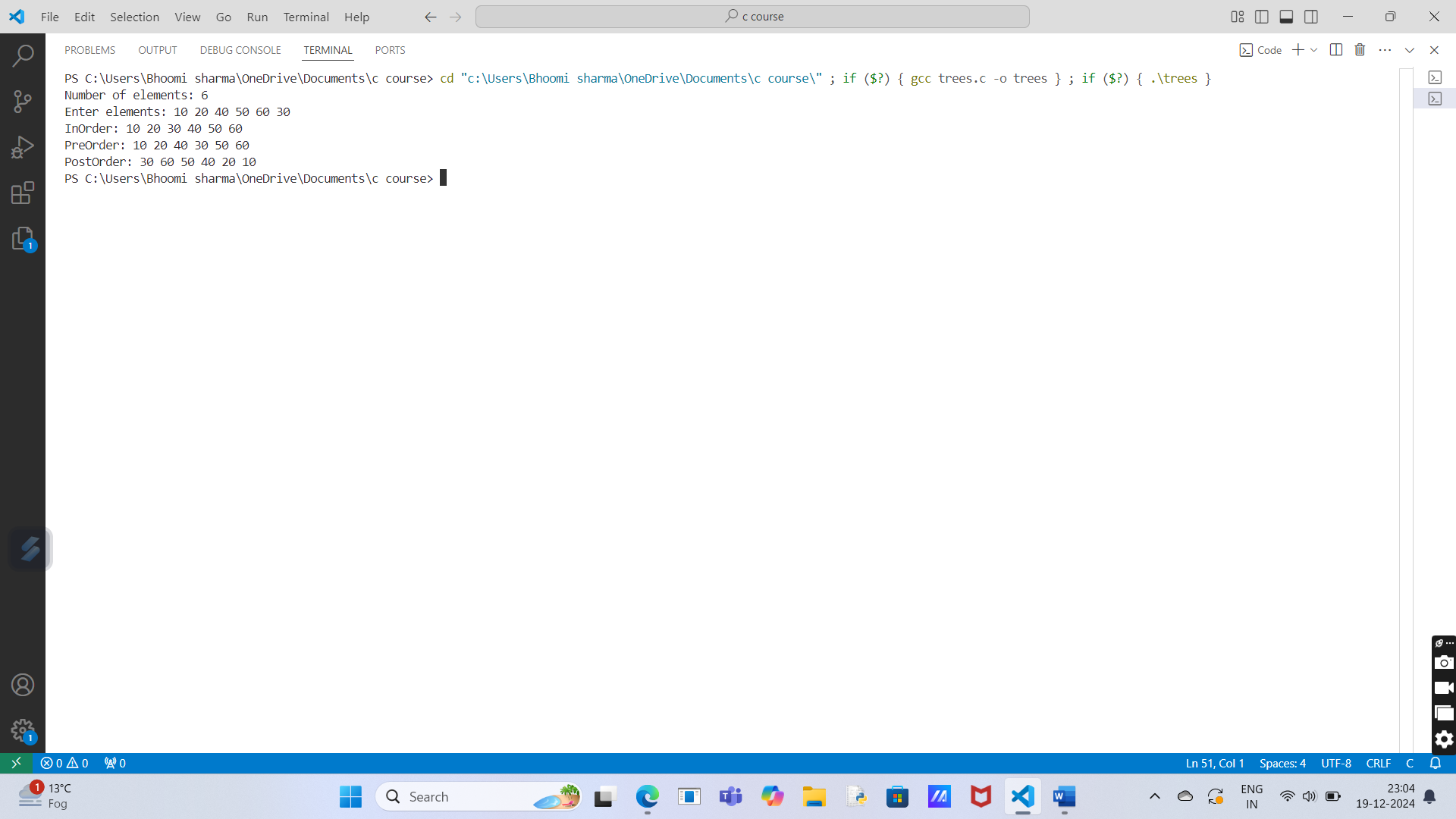
printf("PreOrder: "); preOrder(root); printf("\n");

printf("PostOrder: "); postOrder(root); printf("\n");

return 0;

}

**output:**



**3.Implementation of Kruskal Algorithm**

**Program Description:**

This program implements Kruskal's Algorithm to find the Minimum Spanning Tree (MST) of a graph:

1. Input: Number of vertices, edges, and edge details (source, destination, weight).
2. Process:
   * Sort edges by weight.
   * Use union-find to add edges without forming cycles.
3. Output: Edges included in the MST and their weights.

Kruskal's algorithm ensures the MST is built efficiently using edge sorting and union-find operations.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

// Structure for an edge

struct Edge {

int src, dest, weight;

};

// Structure for a graph

struct Graph {

int V, E;

struct Edge\* edges;

};

// Create a graph

struct Graph\* createGraph(int V, int E) {

struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

graph->V = V;

graph->E = E;

graph->edges = (struct Edge\*)malloc(E \* sizeof(struct Edge));

return graph;

}

// Structure to represent subsets for union-find

struct Subset {

int parent, rank;

};

// Find the set of an element (path compression)

int find(struct Subset subsets[], int i) {

if (subsets[i].parent != i)

subsets[i].parent = find(subsets, subsets[i].parent);

return subsets[i].parent;

}

// Union of two sets by rank

void Union(struct Subset subsets[], int x, int y) {

int rootX = find(subsets, x);

int rootY = find(subsets, y);

if (subsets[rootX].rank < subsets[rootY].rank)

subsets[rootX].parent = rootY;

else if (subsets[rootX].rank > subsets[rootY].rank)

subsets[rootY].parent = rootX;

else {

subsets[rootY].parent = rootX;

subsets[rootX].rank++;

}

}

// Compare edges by weight (for sorting)

int compareEdges(const void\* a, const void\* b) {

struct Edge\* edgeA = (struct Edge\*)a;

struct Edge\* edgeB = (struct Edge\*)b;

return edgeA->weight - edgeB->weight;

}

// Kruskal's Algorithm

void kruskalMST(struct Graph\* graph) {

int V = graph->V;

struct Edge result[V]; // Store the MST

int e = 0; // Index for result

int i = 0; // Index for sorted edges

qsort(graph->edges, graph->E, sizeof(graph->edges[0]), compareEdges);

struct Subset\* subsets = (struct Subset\*)malloc(V \* sizeof(struct Subset));

for (int v = 0; v < V; v++) {

subsets[v].parent = v;

subsets[v].rank = 0;

}

while (e < V - 1 && i < graph->E) {

struct Edge nextEdge = graph->edges[i++];

int x = find(subsets, nextEdge.src);

int y = find(subsets, nextEdge.dest);

if (x != y) {

result[e++] = nextEdge;

Union(subsets, x, y);

}

}

printf("Edges in the Minimum Spanning Tree:\n");

for (i = 0; i < e; i++)

printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);

free(subsets);

}

// Main function

int main() {

int V, E;

printf("Enter the number of vertices and edges: ");

scanf("%d %d", &V, &E);

struct Graph\* graph = createGraph(V, E);

printf("Enter the edges (source, destination, weight):\n");

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

scanf("%d %d %d", &graph->edges[i].src, &graph->edges[i].dest, &graph->edges[i].weight);

}

kruskalMST(graph);

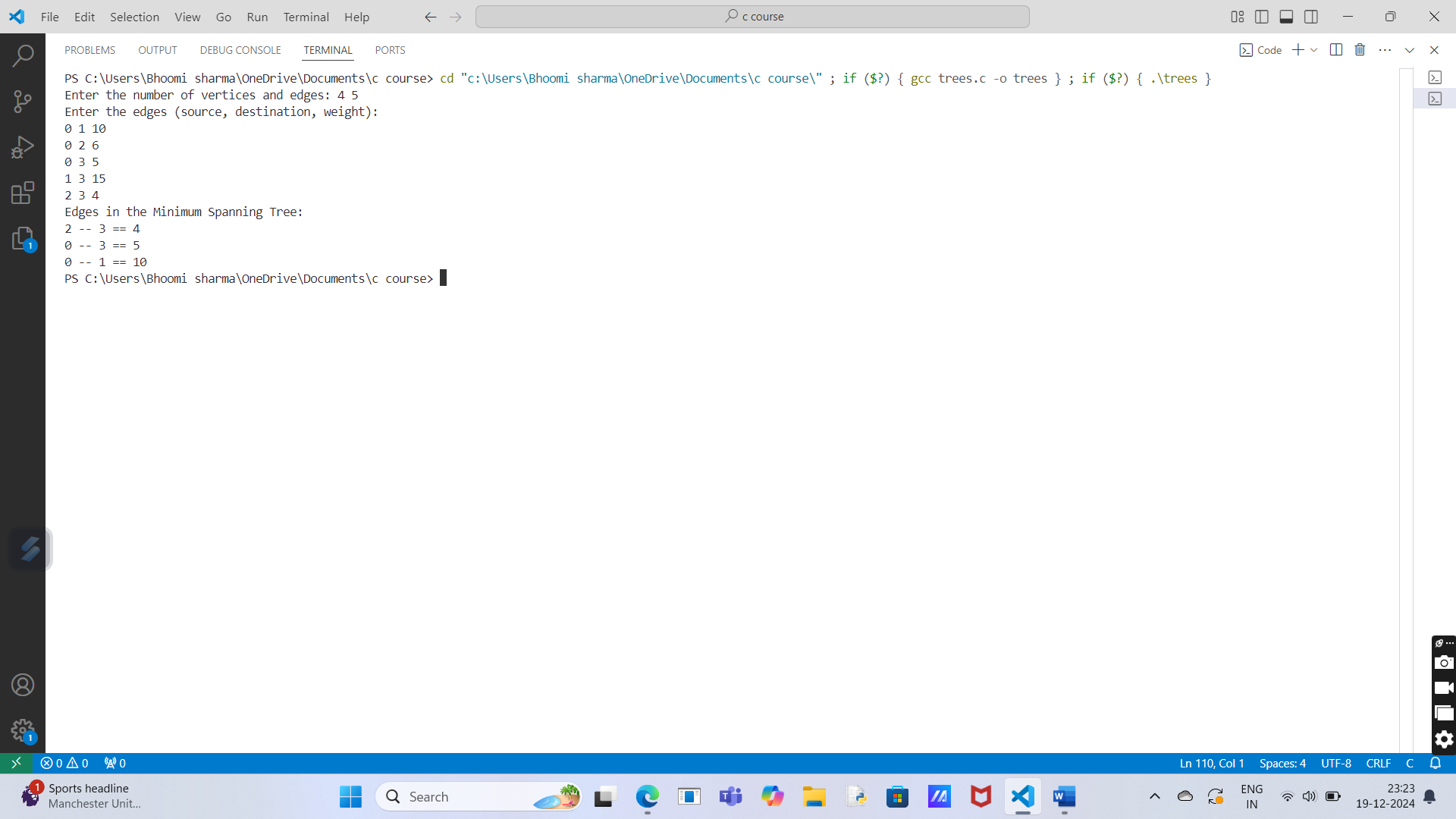
free(graph->edges);

free(graph);

return 0;

**}**

**Output:**



**4.Implementation of Prim Algorithm.**

**Program Description:**

This program implements Prim's Algorithm to find the Minimum Spanning Tree (MST) of a graph:

1. Input: Number of vertices and the adjacency matrix of the graph.
2. Process:
   * Select the smallest edge connecting a visited vertex to an unvisited vertex.
   * Use arrays for parent tracking, key values (minimum edge weight), and visited status.
3. Output: The edges and their weights in the MST**.**

**Solution:**

#include <stdio.h>

#include <limits.h>

#define MAX 100

int findMinVertex(int key[], int visited[], int V) {

int min = INT\_MAX, minIndex = -1;

for (int v = 0; v < V; v++) {

if (!visited[v] && key[v] < min) {

min = key[v];

minIndex = v;

}

}

return minIndex;

}

void primMST(int graph[MAX][MAX], int V) {

int parent[V], key[V], visited[V];

// Initialize arrays

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

key[i] = INT\_MAX;

visited[i] = 0;

}

key[0] = 0; // Start from vertex 0

parent[0] = -1;

for (int count = 0; count < V - 1; count++) {

int u = findMinVertex(key, visited, V);

visited[u] = 1;

for (int v = 0; v < V; v++) {

if (graph[u][v] && !visited[v] && graph[u][v] < key[v]) {

parent[v] = u;

key[v] = graph[u][v];

}

}

}

printf("Edge \tWeight\n");

for (int i = 1; i < V; i++)

printf("%d - %d \t%d\n", parent[i], i, graph[i][parent[i]]);

}

int main() {

int V;

int graph[MAX][MAX];

printf("Enter the number of vertices: ");

scanf("%d", &V);

printf("Enter the adjacency matrix (use 0 for no edge):\n");

for (int i = 0; i < V; i++)

for (int j = 0; j < V; j++)

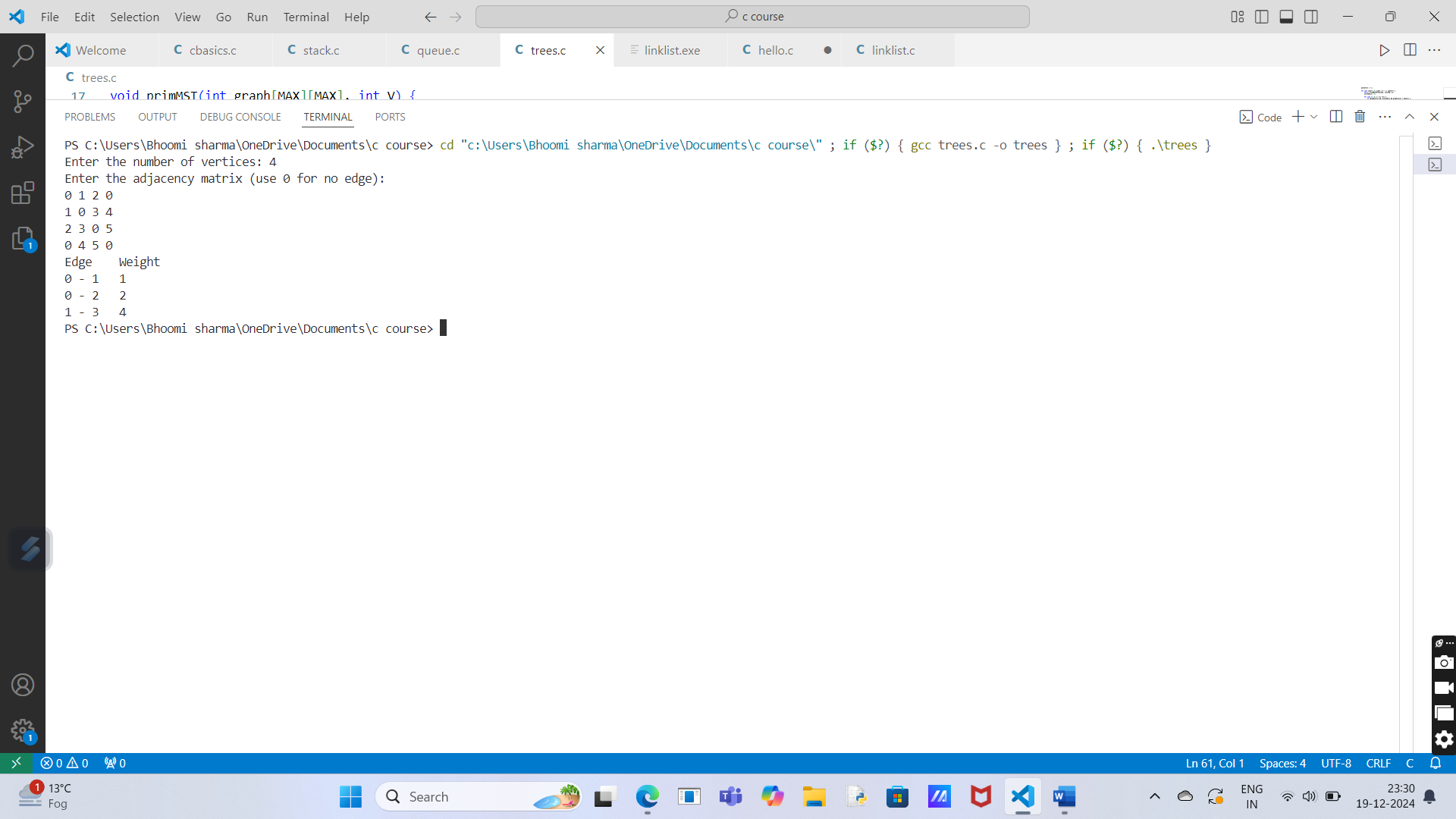
scanf("%d", &graph[i][j]);

primMST(graph, V);

return 0;

}

**Output:**

**5.Implementation of Dijkstra Algorithm**

**Program Description:**

This program implements Dijkstra's Algorithm to find the shortest path from a start vertex to all other vertices in a weighted graph:

1. Input:
   * Number of vertices and the adjacency matrix.
   * Start vertex for the algorithm.
2. Process:
   * Dijkstra’s algorithm is used to calculate the shortest distances from the start vertex.
   * It repeatedly selects the vertex with the smallest tentative distance.
3. Output:
   * Shortest distance from the start vertex to all other vertices.

**Solution:**

#include <stdio.h>

#include <limits.h>

#define MAX 100

// Function to find the vertex with the minimum distance

int findMinVertex(int dist[], int visited[], int V) {

int min = INT\_MAX, minIndex;

for (int v = 0; v < V; v++) {

if (!visited[v] && dist[v] < min) {

min = dist[v];

minIndex = v;

}

}

return minIndex;

}

// Function to implement Dijkstra's Algorithm

void dijkstra(int graph[MAX][MAX], int V, int start) {

int dist[V], visited[V];

// Initialize distances and visited array

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

dist[i] = INT\_MAX;

visited[i] = 0;

}

dist[start] = 0; // Distance to start vertex is 0

// Find shortest path for all vertices

for (int count = 0; count < V - 1; count++) {

int u = findMinVertex(dist, visited, V); // Vertex with min distance

visited[u] = 1;

// Update distance of adjacent vertices of the picked vertex

for (int v = 0; v < V; v++) {

if (graph[u][v] && !visited[v] && dist[u] != INT\_MAX && dist[u] + graph[u][v] < dist[v]) {

dist[v] = dist[u] + graph[u][v];

}

}

}

// Output the shortest distance from start vertex to all others

printf("Vertex\tDistance from Start Vertex %d\n", start);

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

printf("%d \t\t %d\n", i, dist[i]);

}

}

int main() {

int V, start;

int graph[MAX][MAX];

// Input number of vertices and adjacency matrix

printf("Enter the number of vertices: ");

scanf("%d", &V);

printf("Enter the adjacency matrix (0 for no edge):\n");

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

for (int j = 0; j < V; j++) {

scanf("%d", &graph[i][j]);

}

}

// Input the start vertex

printf("Enter the start vertex: ");

scanf("%d", &start);

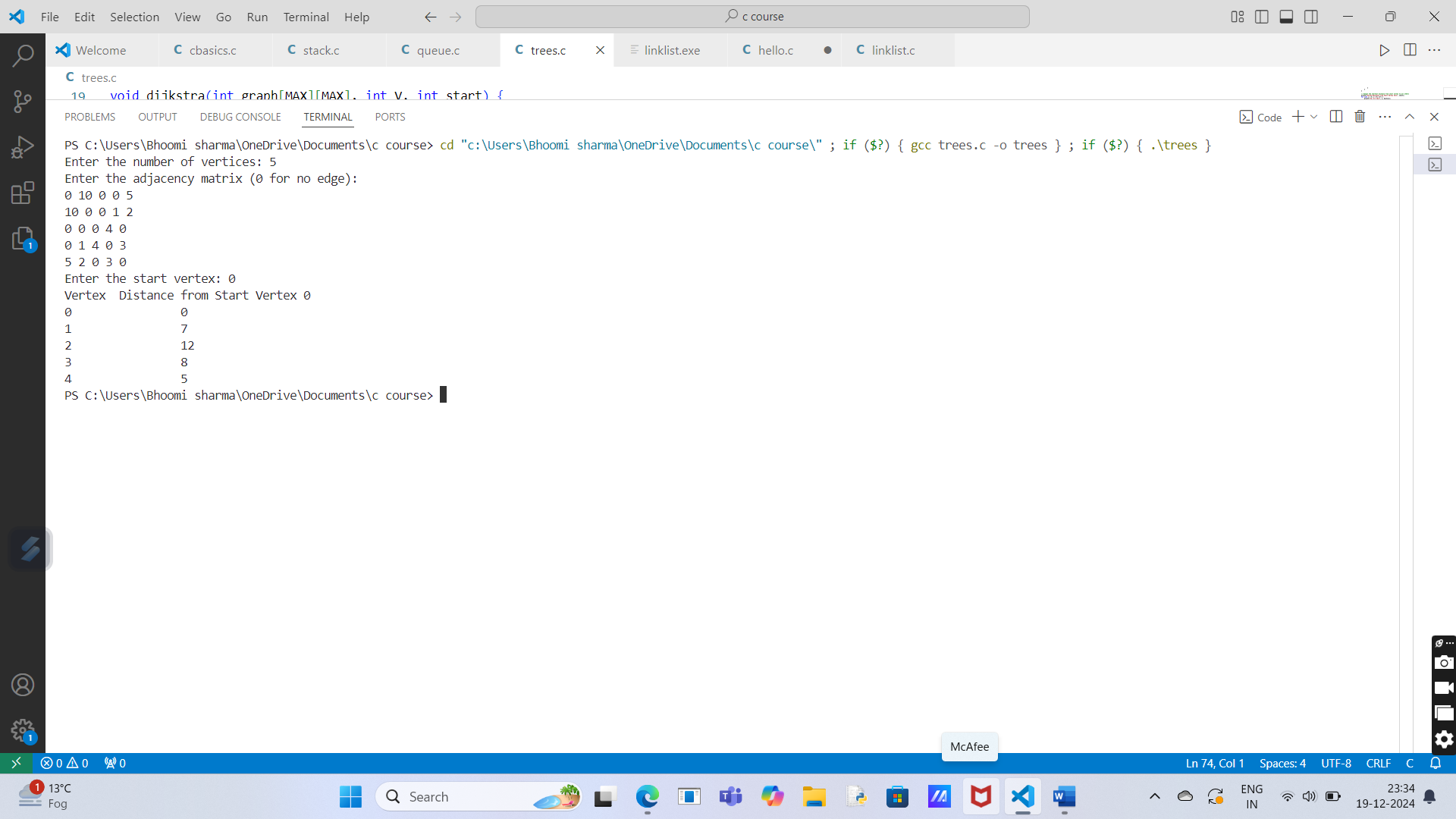
// Call Dijkstra's algorithm

dijkstra(graph, V, start);

return 0;

}

**Output:**



**Section-E (searchind and sorting)**

**Experiment No.: 6**

**1.Implementation of Sorting**

**a.Bubble sort**

**program description:**

The Bubble Sort algorithm sorts an array by repeatedly comparing and swapping adjacent elements if they are in the wrong order. This process continues until no more swaps are needed, meaning the array is sorted.

**Solution:**

#include <stdio.h>

// Function to perform bubble sort

void bubbleSort(int arr[], int n) {

    for (int i = 0; i < n - 1; i++) {

        // Last i elements are already sorted

        for (int j = 0; j < n - i - 1; j++) {

            if (arr[j] > arr[j + 1]) {

                // Swap adjacent elements if they are in the wrong order

                int temp = arr[j];

                arr[j] = arr[j + 1];

                arr[j + 1] = temp;

            }

        }

    }

}

// Function to display the array

void displayArray(int arr[], int n) {

    for (int i = 0; i < n; i++) {

        printf("%d ", arr[i]);

    }

    printf("\n");

}

// Main function

int main() {

    int arr[] = {64, 34, 25, 12, 22, 11, 90};

    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array:\n");

    displayArray(arr, n);

    bubbleSort(arr, n);

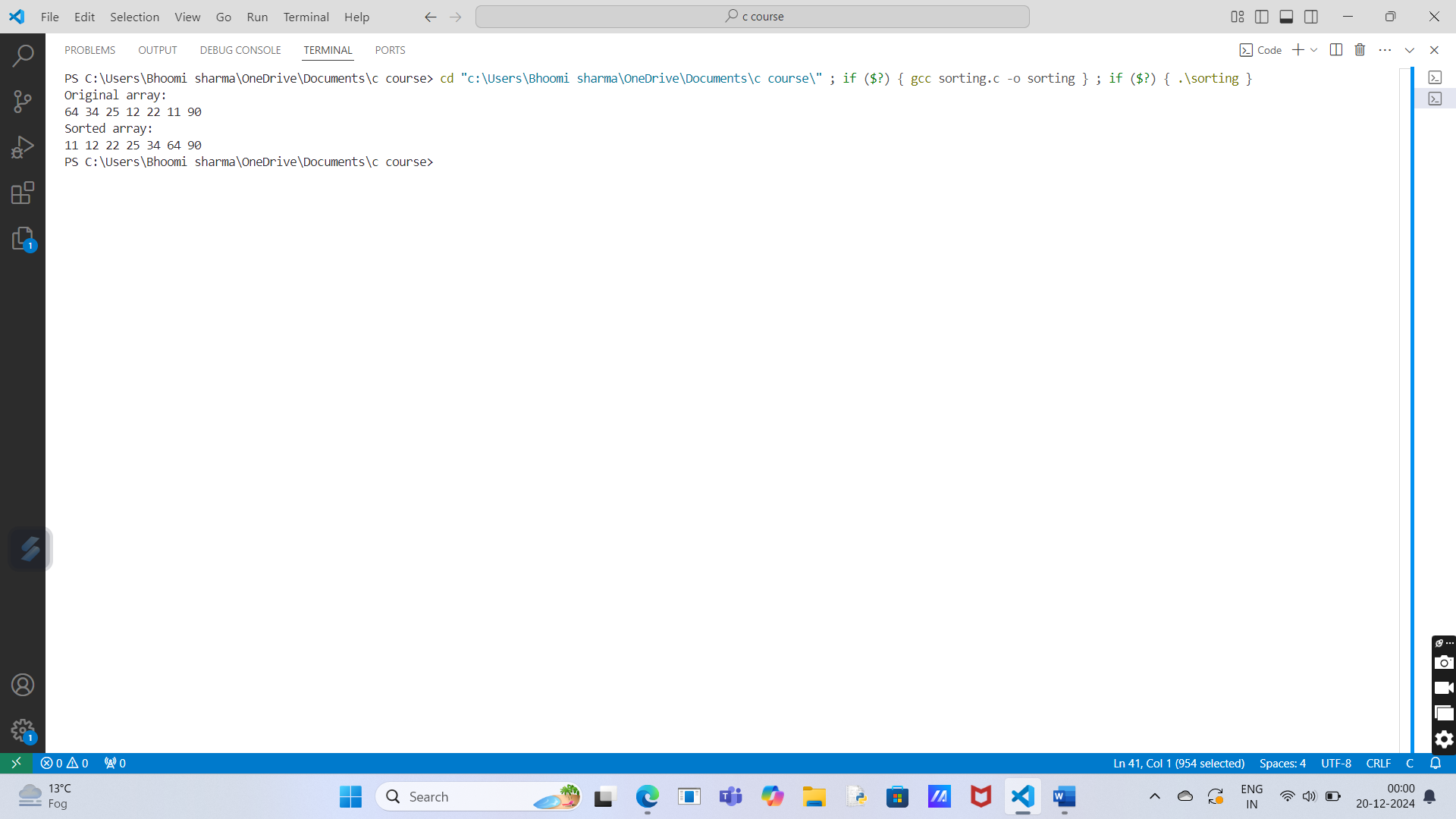
    printf("Sorted array:\n");

    displayArray(arr, n);

    return 0;

}

**Output:**



**B.selection sort.**

**Program Description:**

Selection Sort is a simple sorting algorithm that repeatedly finds the smallest (or largest) element from the unsorted portion of the array and places it at the beginning (or end).

**Solution:**

#include <stdio.h>

// Function to perform selection sort

void selectionSort(int arr[], int n) {

for (int i = 0; i < n - 1; i++) {

int minIndex = i; // Assume the current element is the smallest

for (int j = i + 1; j < n; j++) {

if (arr[j] < arr[minIndex]) {

minIndex = j; // Update minIndex if a smaller element is found

}

}

// Swap the smallest element with the first element of the unsorted part

int temp = arr[minIndex];

arr[minIndex] = arr[i];

arr[i] = temp;

}

}

// Function to display the array

void displayArray(int arr[], int n) {

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

printf("%d ", arr[i]);

}

printf("\n");

}

// Main function

int main() {

int arr[] = {64, 25, 12, 22, 11};

int n = sizeof(arr) / sizeof(arr[0]);

printf("Original array:\n");

displayArray(arr, n);

selectionSort(arr, n);

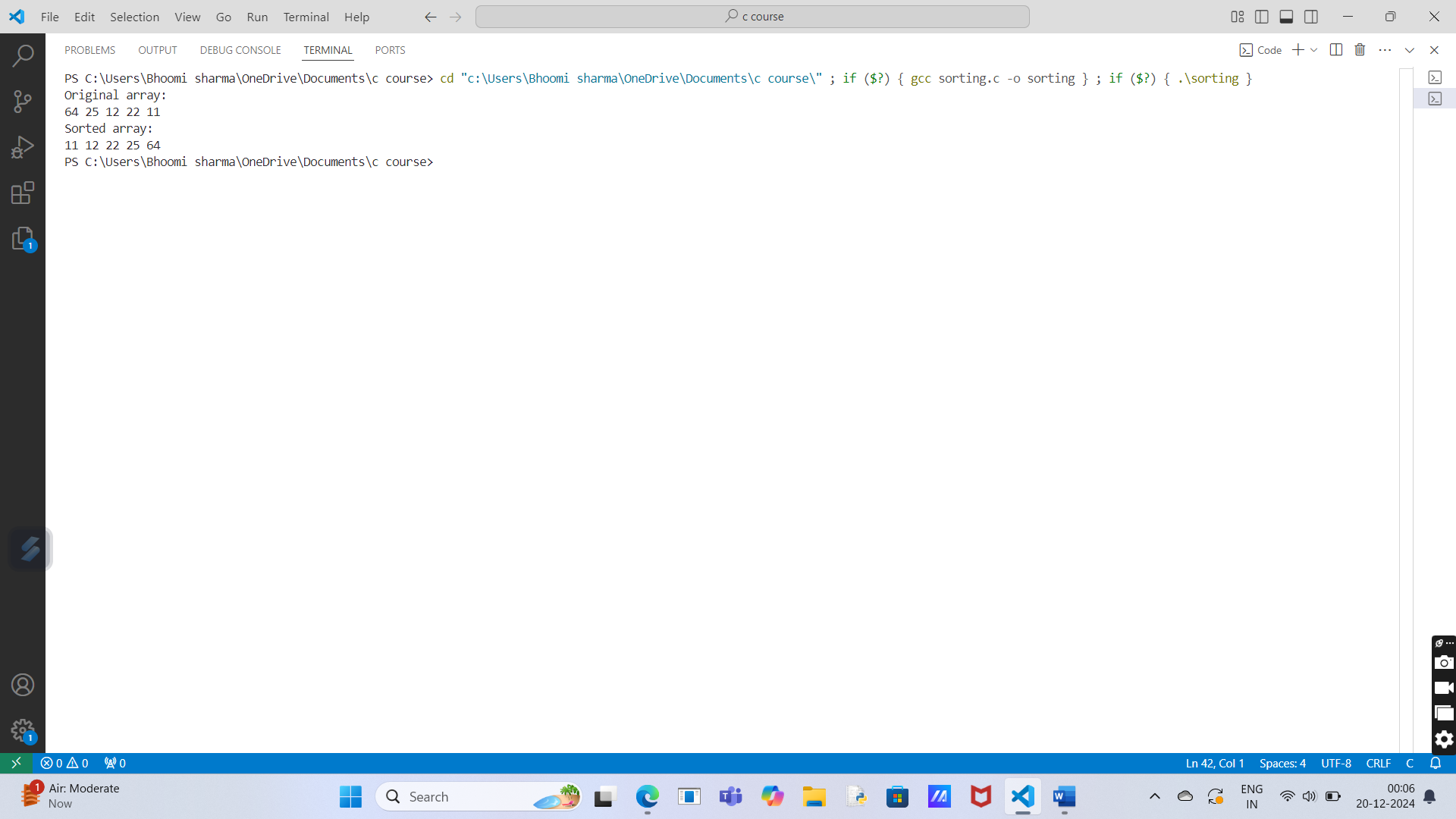
printf("Sorted array:\n");

displayArray(arr, n);

return 0;

}

**Output:**



**C.insertion**

**Program description:**

Insertion Sort is a simple and efficient algorithm for sorting a small number of elements. It works by building the sorted array one element at a time. At each step, it picks the next element from the unsorted part and inserts it into its correct position in the sorted part.

**Solution:**

#include <stdio.h>

// Function to perform insertion sort

void insertionSort(int arr[], int n) {

for (int i = 1; i < n; i++) {

int key = arr[i]; // The element to be inserted

int j = i - 1;

// Move elements of the sorted part to the right

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key; // Insert the key in the correct position

}

}

// Function to display the array

void displayArray(int arr[], int n) {

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

printf("%d ", arr[i]);

}

printf("\n");

}

// Main function

int main() {

int arr[] = {12, 11, 13, 5, 6};

int n = sizeof(arr) / sizeof(arr[0]);

printf("Original array:\n");

displayArray(arr, n);

insertionSort(arr, n);

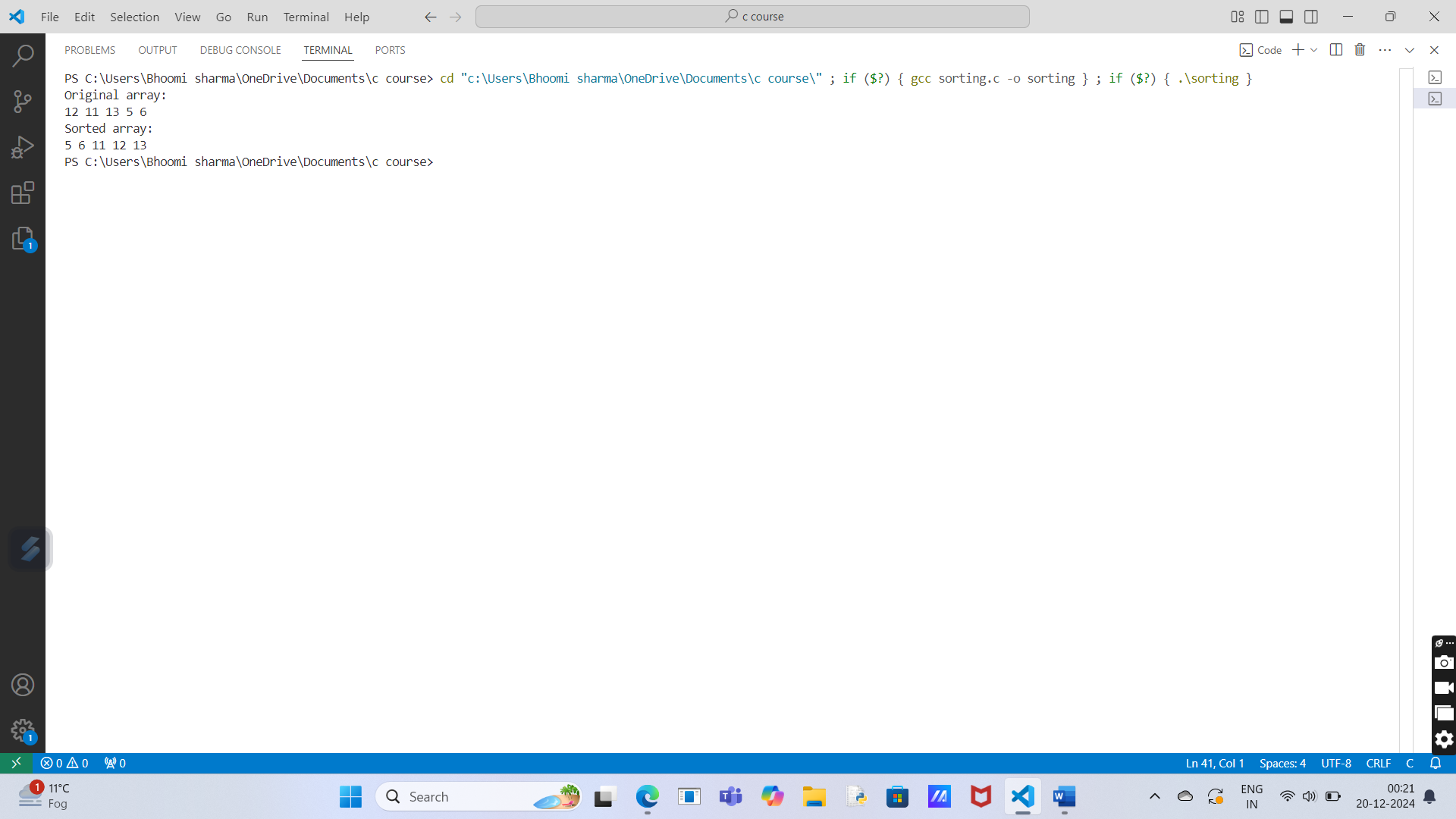
printf("Sorted array:\n");

displayArray(arr, n);

return 0;

}

**Output:**

**D.Quick sort.**

**Program description:**

Quick Sort is a highly efficient sorting algorithm that uses the divide-and-conquer approach. It works by selecting a "pivot" element and partitioning the array into two parts:

1. Elements smaller than the pivot.
2. Elements larger than the pivot.

The process is recursively repeated for the two partitions until the entire array is sorted.

**Solution:**

#include <stdio.h>

// Function to swap two elements

void swap(int \*a, int \*b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to partition the array

int partition(int arr[], int low, int high) {

int pivot = arr[high]; // Choose the last element as the pivot

int i = low - 1; // Index of smaller element

for (int j = low; j < high; j++) {

if (arr[j] < pivot) {

i++;

swap(&arr[i], &arr[j]); // Swap smaller element with the current element

}

}

swap(&arr[i + 1], &arr[high]); // Place the pivot in its correct position

return i + 1; // Return the pivot index

}

// Function to perform Quick Sort

void quickSort(int arr[], int low, int high) {

if (low < high) {

int pi = partition(arr, low, high); // Partition the array

quickSort(arr, low, pi - 1); // Sort the left sub-array

quickSort(arr, pi + 1, high); // Sort the right sub-array

}

}

// Function to display the array

void displayArray(int arr[], int n) {

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

printf("%d ", arr[i]);

}

printf("\n");

}

// Main function

int main() {

int arr[] = {10, 80, 30, 90, 40, 50, 70};

int n = sizeof(arr) / sizeof(arr[0]);

printf("Original array:\n");

displayArray(arr, n);

quickSort(arr, 0, n - 1);

printf("Sorted array:\n");

displayArray(arr, n);

return 0;

}

**Output:**



**E.merge sort.**

**Program description:**

Merge Sort is a divide-and-conquer sorting algorithm. It works by recursively dividing the array into two halves, sorting each half, and then merging the two sorted halves back together.

**Solution:**

#include <stdio.h>

// Function to merge two sub-arrays

void merge(int arr[], int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

// Create temporary arrays

int leftArr[n1], rightArr[n2];

// Copy data to temporary arrays

for (int i = 0; i < n1; i++)

leftArr[i] = arr[left + i];

for (int j = 0; j < n2; j++)

rightArr[j] = arr[mid + 1 + j];

// Merge the temporary arrays back into the original array

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {

if (leftArr[i] <= rightArr[j]) {

arr[k] = leftArr[i];

i++;

} else {

arr[k] = rightArr[j];

j++;

}

k++;

}

// Copy remaining elements of leftArr[] if any

while (i < n1) {

arr[k] = leftArr[i];

i++;

k++;

}

// Copy remaining elements of rightArr[] if any

while (j < n2) {

arr[k] = rightArr[j];

j++;

k++;

}

}

// Function to implement Merge Sort

void mergeSort(int arr[], int left, int right) {

if (left < right) {

int mid = left + (right - left) / 2;

// Recursively sort the two halves

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

// Merge the sorted halves

merge(arr, left, mid, right);

}

}

// Function to display the array

void displayArray(int arr[], int n) {

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

printf("%d ", arr[i]);

}

printf("\n");

}

// Main function

int main() {

int arr[] = {12, 11, 13, 5, 6, 7};

int n = sizeof(arr) / sizeof(arr[0]);

printf("Original array:\n");

displayArray(arr, n);

mergeSort(arr, 0, n - 1);

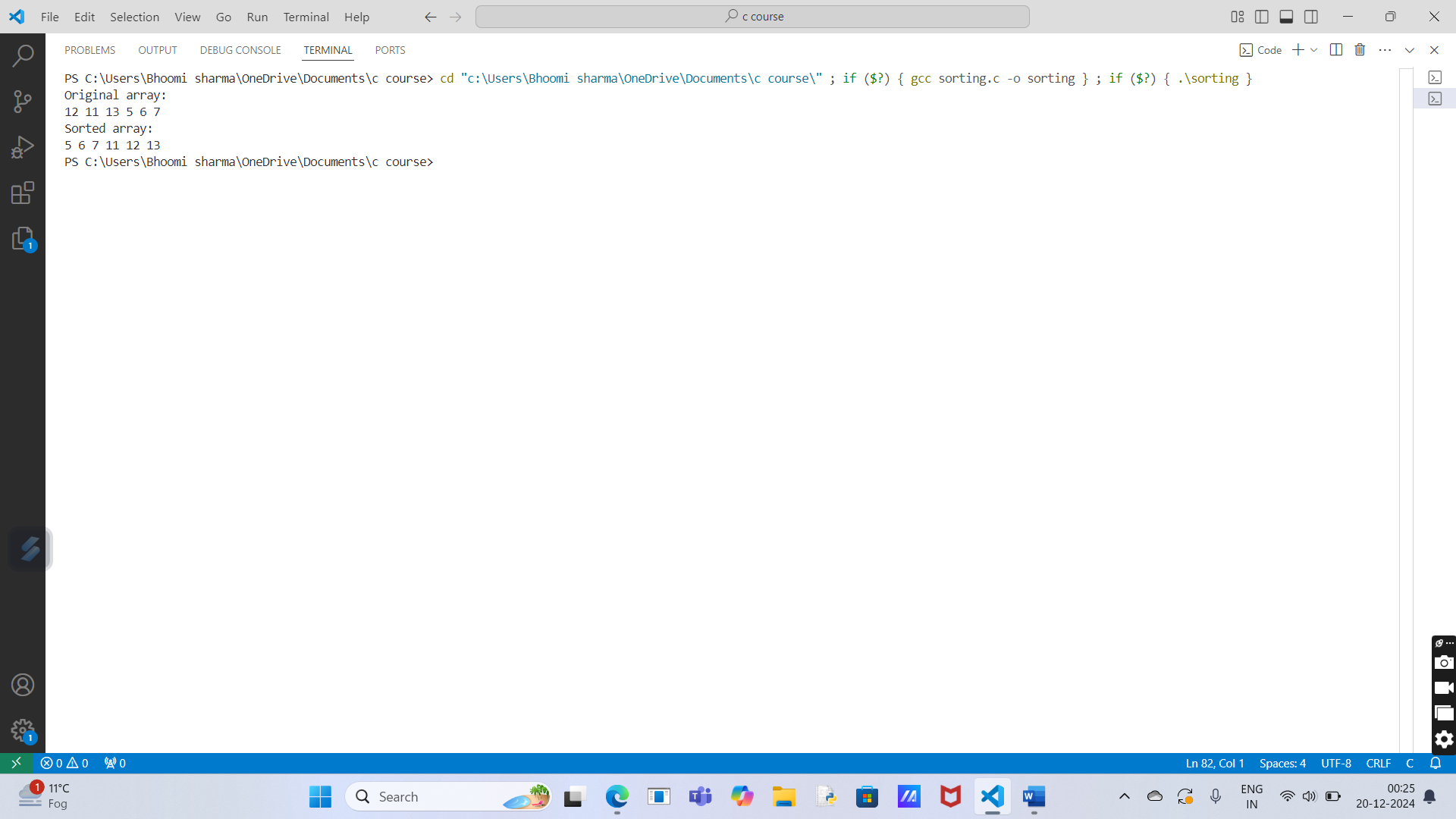
printf("Sorted array:\n");

displayArray(arr, n);

return 0;

}

**Output:**



**2.Implementation of Binary Search on a list of numbers stored in an Array.**

**Program description:**

Binary Search is an efficient searching algorithm that works on sorted arrays. It repeatedly divides the search interval in half:

1. Compare the target value with the middle element of the array.
2. If the target matches the middle element, the search is complete.
3. If the target is smaller, narrow the search to the left half; otherwise, search the right half.
4. Repeat until the target is found or the search interval is empty.

**Solution:**

#include <stdio.h>

// Function to perform binary search

int binarySearch(int arr[], int n, int target) {

int left = 0, right = n - 1;

while (left <= right) {

int mid = left + (right - left) / 2; // Calculate mid-point

// Check if the target is at the mid-point

if (arr[mid] == target)

return mid;

// If the target is smaller, search the left half

if (arr[mid] > target)

right = mid - 1;

else

left = mid + 1; // Otherwise, search the right half

}

// Target not found

return -1;

}

// Function to display the result

void displayResult(int result) {

if (result != -1)

printf("Element found at index: %d\n", result);

else

printf("Element not found in the array.\n");

}

// Main function

int main() {

int arr[] = {2, 3, 4, 10, 40};

int n = sizeof(arr) / sizeof(arr[0]);

int target = 10;

printf("Array: ");

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

printf("%d ", arr[i]);

}

printf("\nSearching for %d...\n", target);

int result = binarySearch(arr, n, target);

displayResult(result);

return 0;

}

**Output:**



**3. Implementation of Binary Search on a list of strings stored in an Array**

**Program descripton:**

Binary Search can also be applied to a list of strings, provided the list is sorted alphabetically. The logic is the same as numeric Binary Search, but instead of numeric comparisons, we use string comparisons.

**Solution:**

#include <stdio.h>

#include <string.h>

// Function to perform binary search on a list of strings

int binarySearchStrings(char arr[][20], int n, char target[]) {

int left = 0, right = n - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

int res = strcmp(arr[mid], target);

// Check if the target is at the mid-point

if (res == 0)

return mid;

// If the target is lexicographically smaller, search the left half

if (res > 0)

right = mid - 1;

else

left = mid + 1; // Otherwise, search the right half

}

// Target not found

return -1;

}

// Main function

int main() {

char arr[][20] = {"apple", "banana", "cherry", "date", "fig", "grape"};

int n = sizeof(arr) / sizeof(arr[0]);

char target[20] = "cherry";

printf("Array of strings:\n");

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

printf("%s ", arr[i]);

}

printf("\nSearching for \"%s\"...\n", target);

int result = binarySearchStrings(arr, n, target);

if (result != -1)

printf("String found at index: %d\n", result);

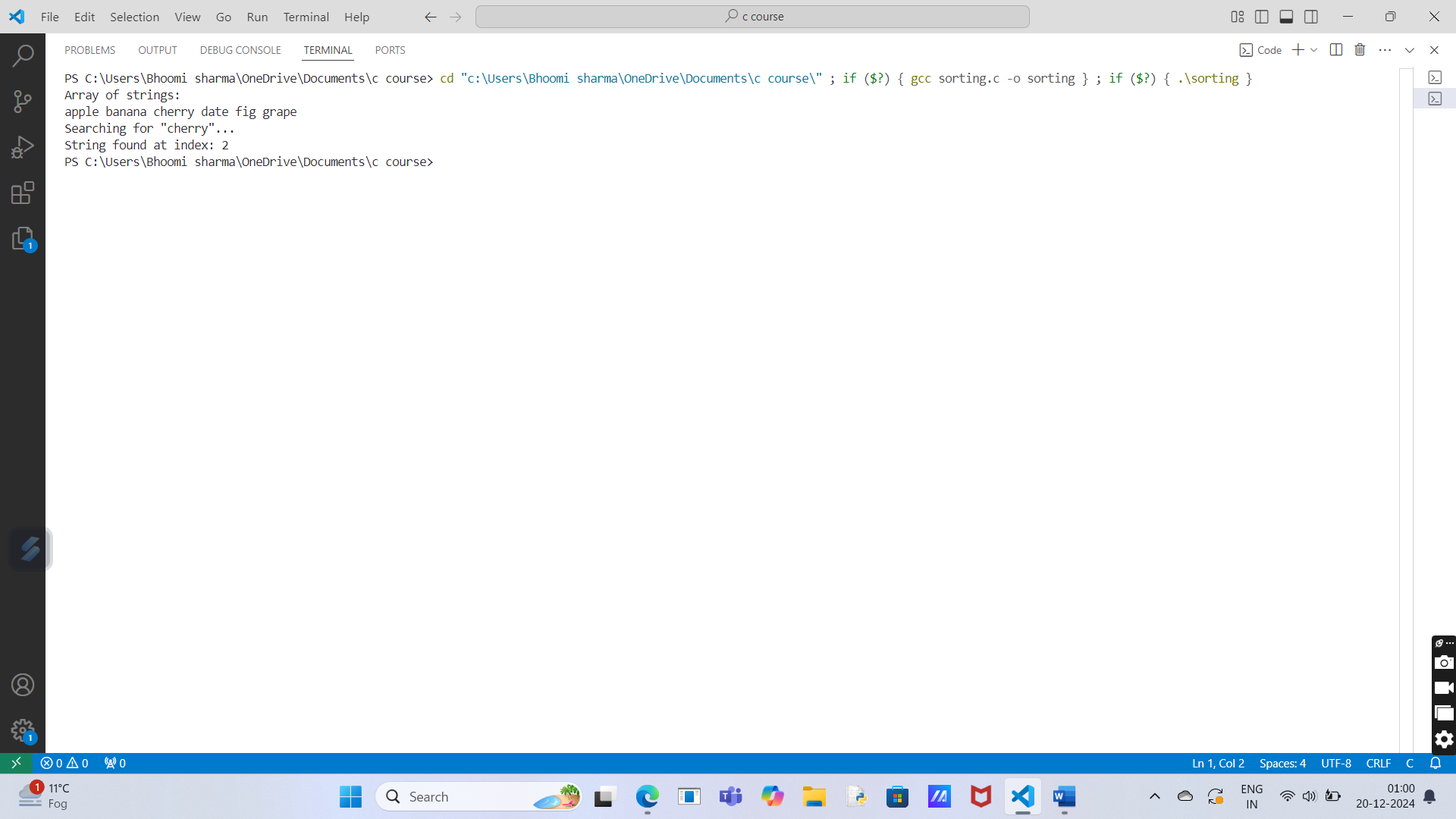
else

printf("String not found in the array.\n");

return 0;

}

**Output:**



**4. Implementation of Linear Search on a list of strings stored in an Array**

**Program description:**

Linear Search is a simple searching algorithm that works by checking each element in an array one by one until the target string is found or the end of the array is reached.

This algorithm does not require the array to be sorted and is suitable for small datasets.

**Solution:**

#include <stdio.h>

#include <string.h>

// Function to perform linear search on a list of strings

int linearSearchStrings(char arr[][20], int n, char target[]) {

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

if (strcmp(arr[i], target) == 0) {

return i; // Return the index if the string matches

}

}

return -1; // Return -1 if the string is not found

}

// Main function

int main() {

char arr[][20] = {"apple", "banana", "cherry", "date", "fig", "grape"};

int n = sizeof(arr) / sizeof(arr[0]);

char target[20] = "date";

printf("Array of strings:\n");

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

printf("%s ", arr[i]);

}

printf("\nSearching for \"%s\"...\n", target);

int result = linearSearchStrings(arr, n, target);

if (result != -1)

printf("String found at index: %d\n", result);

else

printf("String not found in the array.\n");

return 0;

}

**Output:**

