

**Galgotias College of Engineering and Technology**

Master of Computer Application

II Semester

2023 - 2024

Data Structure and Analysis of Algorithm Lab

**Submitted by:** Akash Yadav 2300970140019

**Program: - 1**

**Objective:** Write a Program to implement addition and multiplication in Two Dimension Array.

**Code:**

#include <stdio.h>

#define ROWS 3

#define COLS 3

void addMatrices(int matrix1[ROWS][COLS], int matrix2[ROWS][COLS], int result[ROWS][COLS]) {

    for (int i = 0; i < ROWS; i++) {

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

            result[i][j] = matrix1[i][j] + matrix2[i][j];

        }

    }

}

void multiplyMatrices(int matrix1[ROWS][COLS], int matrix2[ROWS][COLS], int result[ROWS][COLS]) {

    for (int i = 0; i < ROWS; i++) {

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

            result[i][j] = 0;

            for (int k = 0; k < COLS; k++) {

                result[i][j] += matrix1[i][k] \* matrix2[k][j];

            }

        }

    }

}

void printMatrix(int matrix[ROWS][COLS]) {

    for (int i = 0; i < ROWS; i++) {

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

            printf("%d ", matrix[i][j]);

        }

        printf("\n");

    }

}

int main() {

    int matrix1[ROWS][COLS] = {

        {1, 2, 3},

        {4, 5, 6},

        {7, 8, 9}

    };

    int matrix2[ROWS][COLS] = {

        {9, 8, 7},

        {6, 5, 4},

        {3, 2, 1}

    };

    int result\_addition[ROWS][COLS];

    int result\_multiplication[ROWS][COLS];

    printf("Matrix 1:\n");

    printMatrix(matrix1);

    printf("\nMatrix 2:\n");

    printMatrix(matrix2);

*// Addition*

    printf("\nAddition:\n");

    addMatrices(matrix1, matrix2, result\_addition);

    printMatrix(result\_addition);

*// Multiplication*

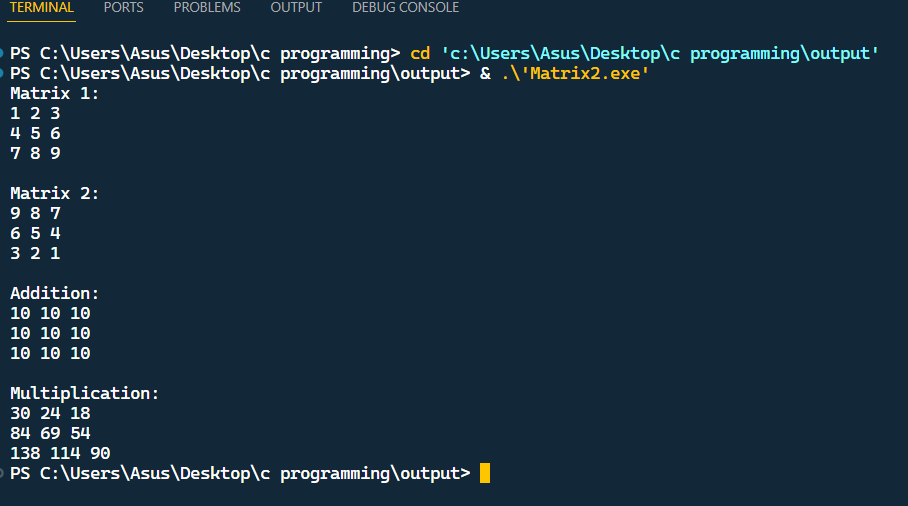
    printf("\nMultiplication:\n");

    multiplyMatrices(matrix1, matrix2, result\_multiplication);

    printMatrix(result\_multiplication);

    return 0;

}

**Output:-**

**Program: -2**

**Objective: -**Write a program to transpose a 2-d Array.

**Code: -**

#include <stdio.h>

#define ROWS 3

#define COLS 3

void transpose(int matrix[ROWS][COLS], int result[COLS][ROWS]) {

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

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

result[j][i] = matrix[i][j];

}

}

}

void printMatrix(int matrix[ROWS][COLS]) {

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

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

printf("%d ", matrix[i][j]);

}

printf("\n");

}

}

int main() {

int matrix[ROWS][COLS] = {

{1, 2, 3},

{4, 5, 6},

{7, 8, 9}

};

int result[COLS][ROWS];

printf("Original Matrix:\n");

printMatrix(matrix);

transpose(matrix, result);

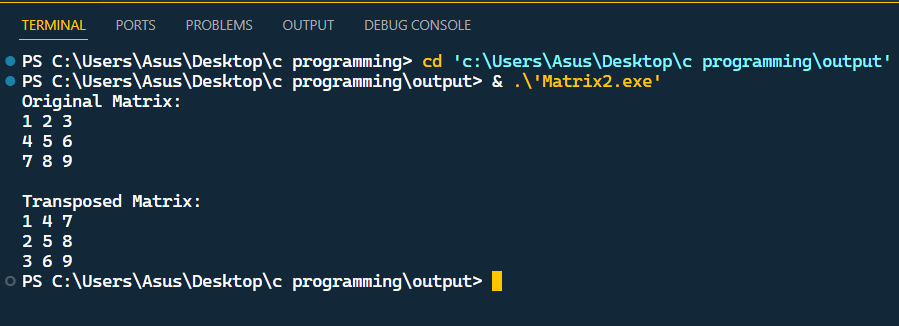
printf("\nTransposed Matrix:\n");

printMatrix(result);

return 0;

}

**Output: -**

****

**Program: -3**

**Objective: -** Write a program to implement a stack using array.

**Code: -**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_SIZE 100

typedef struct {

    int arr[MAX\_SIZE];

    int top;

} *Stack*;

void initializeStack(*Stack* \*stack) {

    stack->top = -1; *// Set top to -1 to indicate an empty stack*

}

bool isEmpty(*Stack* \*stack) {

    return stack->top == -1;

}

bool isFull(*Stack* \*stack) {

    return stack->top == MAX\_SIZE - 1;

}

void push(*Stack* \*stack, int element) {

    if (isFull(stack)) {

        printf("Error: Stack is full, cannot push element.\n");

        return;

    }

    stack->top++;

    stack->arr[stack->top] = element;

}

int pop(*Stack* \*stack) {

    if (isEmpty(stack)) {

        printf("Error: Stack is empty, cannot pop element.\n");

        return -1; *// Return -1 to indicate an empty stack*

    }

    int poppedElement = stack->arr[stack->top];

    stack->top--;

    return poppedElement;

}

int peek(*Stack* \*stack) {

    if (isEmpty(stack)) {

        printf("Error: Stack is empty, cannot peek.\n");

        return -1; *// Return -1 to indicate an empty stack*

    }

    return stack->arr[stack->top];

}

int main() {

*Stack* stack;

    initializeStack(&stack);

    push(&stack, 10);

    push(&stack, 20);

    push(&stack, 30);

    printf("Top element of the stack: %d\n", peek(&stack));

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

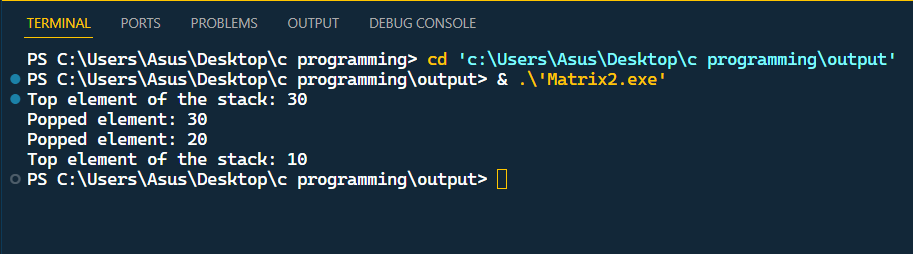
    printf("Popped element: %d\n", pop(&stack));

    printf("Top element of the stack: %d\n", peek(&stack));

    return 0;

}

**Output: -**

****

**Program:-4**

**Objective:-**Write a program to implement Queue using array.

**Code:-**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

typedef struct {

    int items[MAX\_SIZE];

    int front;

    int rear;

} *Queue*;

*// Function prototypes*

void initializeQueue(*Queue* \*q);

int isEmpty(*Queue* \*q);

int isFull(*Queue* \*q);

void enqueue(*Queue* \*q, int value);

int dequeue(*Queue* \*q);

void displayQueue(*Queue* \*q);

*// Initialize the queue*

void initializeQueue(*Queue* \*q) {

    q->front = -1;

    q->rear = -1;

}

*// Check if the queue is empty*

int isEmpty(*Queue* \*q) {

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

}

*// Check if the queue is full*

int isFull(*Queue* \*q) {

    return (q->rear + 1) % MAX\_SIZE == q->front ? 1 : 0;

}

*// Add an element to the queue*

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

    if (isFull(q)) {

        printf("Queue is full, cannot enqueue.\n");

        return;

    } else if (isEmpty(q)) {

        q->front = 0;

        q->rear = 0;

    } else {

        q->rear = (q->rear + 1) % MAX\_SIZE;

    }

    q->items[q->rear] = value;

}

*// Remove an element from the queue*

int dequeue(*Queue* \*q) {

    int value;

    if (isEmpty(q)) {

        printf("Queue is empty, cannot dequeue.\n");

        return -1;

    } else if (q->front == q->rear) {

        value = q->items[q->front];

        q->front = -1;

        q->rear = -1;

    } else {

        value = q->items[q->front];

        q->front = (q->front + 1) % MAX\_SIZE;

    }

    return value;

}

*// Display the elements of the queue*

void displayQueue(*Queue* \*q) {

    if (isEmpty(q)) {

        printf("Queue is empty.\n");

        return;

    }

    printf("Queue: ");

    int i = q->front;

    while (i != q->rear) {

        printf("%d ", q->items[i]);

        i = (i + 1) % MAX\_SIZE;

    }

    printf("%d\n", q->items[q->rear]);

}

*// Main function*

int main() {

*Queue* q;

    initializeQueue(&q);

*// Example usage*

    enqueue(&q, 10);

    enqueue(&q, 20);

    enqueue(&q, 30);

    displayQueue(&q);

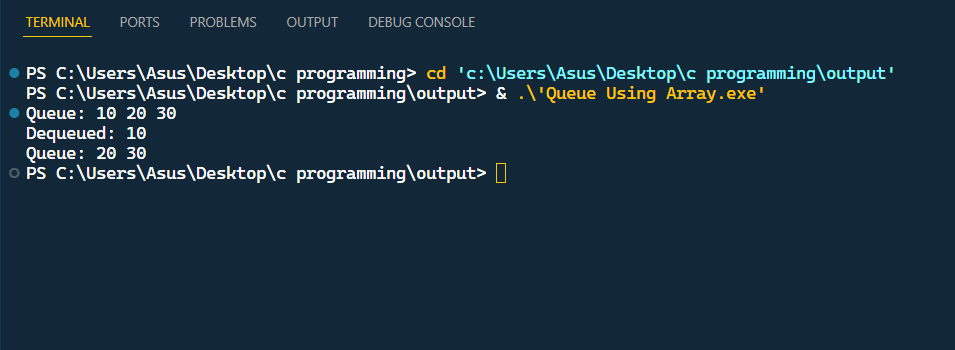
    printf("Dequeued: %d\n", dequeue(&q));

    displayQueue(&q);

    return 0;

}

**Output:-**

****

**Program:-5**

**Object:-**Write a Program to implement a circular queue using array.

**Code:-** *// Circular Queue implementation in C*

#include <stdio.h>

#define SIZE 5

int items[SIZE];

int front = -1, rear = -1;

*// Check if the queue is full*

int isFull() {

  if ((front == rear + 1) || (front == 0 && rear == SIZE - 1)) return 1;

  return 0;

}

*// Check if the queue is empty*

int isEmpty() {

  if (front == -1) return 1;

  return 0;

}

*// Adding an element*

void enQueue(int element) {

  if (isFull())

    printf("\n Queue is full!! \n");

  else {

    if (front == -1) front = 0;

    rear = (rear + 1) % SIZE;

    items[rear] = element;

    printf("\n Inserted -> %d", element);

  }

}

*// Removing an element*

int deQueue() {

  int element;

  if (isEmpty()) {

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

    return (-1);

  } else {

    element = items[front];

    if (front == rear) {

      front = -1;

      rear = -1;

    }

*// Q has only one element, so we reset the*

*// queue after dequeing it. ?*

    else {

      front = (front + 1) % SIZE;

    }

    printf("\n Deleted element -> %d \n", element);

    return (element);

  }

}

*// Display the queue*

void display() {

  int i;

  if (isEmpty())

    printf(" \n Empty Queue\n");

  else {

    printf("\n Front -> %d ", front);

    printf("\n Items -> ");

    for (i = front; i != rear; i = (i + 1) % SIZE) {

      printf("%d ", items[i]);

    }

    printf("%d ", items[i]);

    printf("\n Rear -> %d \n", rear);

  }

}

int main() {

*// Fails because front = -1*

  deQueue();

  enQueue(1);

  enQueue(2);

  enQueue(3);

  enQueue(4);

  enQueue(5);

*// Fails to enqueue because front == 0 && rear == SIZE - 1*

  enQueue(6);

  display();

  deQueue();

  display();

  enQueue(7);

  display();

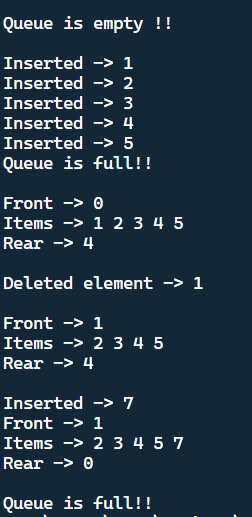
*// Fails to enqueue because front == rear + 1*

  enQueue(8);

  return 0;

}

**Output:-**

****

**Program:-6**

**Object:-**Write a program to implement stack using linkedlist.

**Code:-** *// C program to implement a stack using linked list*

#include <stdio.h>

#include <stdlib.h>

*//  \_\_\_\_\_\_\_\_LINKED LIST UTILITY FUNCITON\_\_\_\_\_\_\_\_\_\_\_\_*

*// Define the structure for a node of the linked list*

typedef struct *Node*

{

    int data;

    struct *Node* \*next;

} *node*;

*// linked list utility function*

*node* \*createNode(int data)

{

*// allocating memory*

*node* \*newNode = (*node* \*)malloc(sizeof(*node*));

*// if memory allocation is failed*

    if (newNode == NULL)

        return NULL;

*// putting data in the node*

    newNode->data = data;

    newNode->next = NULL;

    return newNode;

}

*// fuction to insert data before the head node*

int insertBeforeHead(*node* \*\*head, int data)

{

*// creating new node*

*node* \*newNode = createNode(data);

*// if malloc fail, return error code*

    if (!newNode)

        return -1;

*// if the linked list is empty*

    if (\*head == NULL)

    {

        \*head = newNode;

        return 0;

    }

    newNode->next = \*head;

    \*head = newNode;

    return 0;

}

*// deleting head node*

int deleteHead(*node* \*\*head)

{

*// no need to check for empty stack as it is already*

*// being checked in the caller function*

*node* \*temp = \*head;

    \*head = (\*head)->next;

    free(temp);

    return 0;

}

*// \_\_\_\_\_\_\_\_\_STACK IMPLEMENTATION STARTS HERE\_\_\_\_\_\_\_\_\_*

*// Function to check if the stack is empty or not*

int isEmpty(*node* \*\*stack) { return \*stack == NULL; }

*// Function to push elements to the stack*

void push(*node* \*\*stack, int data)

{

*// inserting the data at the beginning of the linked*

*// list stack*

*// if the insertion function returns the non - zero*

*// value, it is the case of stack overflow*

    if (insertBeforeHead(stack, data))

    {

        printf("Stack Overflow!\n");

    }

}

*// Function to pop an element from  the stack*

int pop(*node* \*\*stack)

{

*// checking underflow condition*

    if (isEmpty(stack))

    {

        printf("Stack Underflow\n");

        return -1;

    }

*// deleting the head.*

    deleteHead(stack);

}

*// Function to return the topmost element of the stack*

int peek(*node* \*\*stack)

{

*// check for empty stack*

    if (!isEmpty(stack))

        return (\*stack)->data;

    else

        return -1;

}

*// Function to print the Stack*

void printStack(*node* \*\*stack)

{

*node* \*temp = \*stack;

    while (temp != NULL)

    {

        printf("%d-> ", temp->data);

        temp = temp->next;

    }

    printf("\n");

}

*// driver code*

int main()

{

*// Initialize a new stack top pointer*

*node* \*stack = NULL;

*// Push elements into the stack*

    push(&stack, 10);

    push(&stack, 20);

    push(&stack, 30);

    push(&stack, 40);

    push(&stack, 50);

*// Print the stack*

    printf("Stack: ");

    printStack(&stack);

*// Pop elements from the stack*

    pop(&stack);

    pop(&stack);

*// Print the stack after deletion of elements*

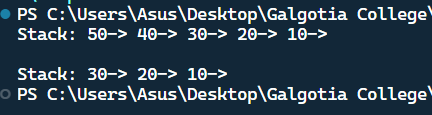
    printf("\nStack: ");

    printStack(&stack);

    return 0;

}

**Output:-**

****

**Program:-7**

**Object:-**Write program to implement queue using a linkedlist.

**Code:-**

#include <stdio.h>

#include <stdlib.h>

*// Node structure for linked list*

typedef struct *Node* {

    int data;

    struct *Node* \*next;

} *Node*;

*// Queue structure with pointers to front and rear nodes*

typedef struct {

*Node* \*front;

*Node* \*rear;

} *Queue*;

*// Function to initialize the queue*

void initializeQueue(*Queue* \*q) {

    q->front = NULL;

    q->rear = NULL;

}

*// Function to check if the queue is empty*

int isEmpty(*Queue* \*q) {

    return (q->front == NULL);

}

*// Function to enqueue (add) an element to the queue*

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

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

    if (!newNode) {

        printf("Memory allocation error\n");

        return;

    }

    newNode->data = value;

    newNode->next = NULL;

    if (q->rear) {

        q->rear->next = newNode;

    }

    q->rear = newNode;

    if (!q->front) {

        q->front = newNode;

    }

    printf("Enqueued %d\n", value);

}

*// Function to dequeue (remove) an element from the queue*

int dequeue(*Queue* \*q) {

    if (isEmpty(q)) {

        printf("Queue Underflow\n");

        return -1;

    }

*Node* \*temp = q->front;

    int value = temp->data;

    q->front = q->front->next;

    if (!q->front) {

        q->rear = NULL;

    }

    free(temp);

    return value;

}

*// Function to display the queue's contents*

void display(*Queue* \*q) {

    if (isEmpty(q)) {

        printf("Queue is empty\n");

        return;

    }

*Node* \*temp = q->front;

    printf("Queue elements: ");

    while (temp) {

        printf("%d ", temp->data);

        temp = temp->next;

    }

    printf("\n");

}

int main() {

*Queue* q;

    initializeQueue(&q);

    enqueue(&q, 10);

    enqueue(&q, 20);

    enqueue(&q, 30);

    display(&q);

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

    display(&q);

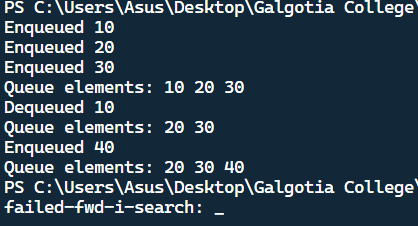
    enqueue(&q, 40);

    display(&q);

    return 0;

}

**Output:-**

****

**Program:-8**

**Object:-** Write a program to implement BFS using linkedlist

**Code:-**

#include <stdio.h>

#include <stdlib.h>

struct *node*

{

    int data;

    struct *node* \*left;

    struct *node* \*right;

};

struct *queue*

{

    int size;

    int front;

    int rear;

    struct *node* \*\*Q;

};

void createQueue(struct *queue* \*q, int size)

{

    q->size = size;

    q->front = q->rear = -1;

    q->Q = (struct *node* \*\*)malloc(q->size \* sizeof(struct *node* \*));

}

void enqueue(struct *queue* \*q, struct *node* \*x)

{

    if (q->rear == q->size - 1)

        printf("Queue is Full");

    else

    {

        q->rear++;

        q->Q[q->rear] = x;

    }

}

struct *node* \*dequeue(struct *queue* \*q)

{

    if (q->front == q->rear)

        printf("Queue is Empty");

    else

    {

        q->front++;

        return q->Q[q->front];

    }

}

void BFS(struct *node* \*root)

{

    struct *queue* q;

    createQueue(&q, 100);

    struct *node* \*temp;

    enqueue(&q, root);

    while (q.front != q.rear)

    {

        temp = dequeue(&q);

        printf("%d ", temp->data);

        if (temp->left != NULL)

            enqueue(&q, temp->left);

        if (temp->right != NULL)

            enqueue(&q, temp->right);

    }

}

int main()

{

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

    root->data = 1;

    root->left = (struct *node* \*)malloc(sizeof(struct *node*));

    root->left->data = 2;

    root->right = (struct *node* \*)malloc(sizeof(struct *node*));

    root->right->data = 3;

    root->left->left = (struct *node* \*)malloc(sizeof(struct *node*));

    root->left->left->data = 4;

    root->left->right = (struct *node* \*)malloc(sizeof(struct *node*));

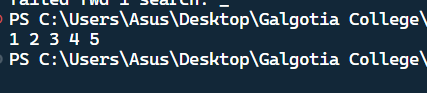
    root->left->right->data = 5;

    BFS(root);

    return 0;

}

**Output:-**

****

**Program:-9**

**Object:-** Write a program to implement DFS using linkedlist.

**Code:-**

#include <stdio.h>

#include <stdlib.h>

struct *node*

{

    int data;

    struct *node* \*left;

    struct *node* \*right;

};

void preorderDfs(struct *node* \*root)

{

    if (root == NULL)

    {

        return;

    }

    printf("%d-> ", root->data);

    preorderDfs(root->left);

    preorderDfs(root->right);

}

void inorderDfs(struct *node* \*root)

{

    if (root == NULL)

    {

        return;

    }

    inorderDfs(root->left);

    printf("%d->", root->data);

    inorderDfs(root->right);

}

void postorderDfs(struct *node* \*root)

{

    if (root == NULL)

    {

        return;

    }

    postorderDfs(root->left);

    postorderDfs(root->right);

    printf("%d->", root->data);

}

int main()

{

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

    root->data = 1;

    root->left = (struct *node* \*)malloc(sizeof(struct *node*));

    root->left->data = 2;

    root->left->left = NULL;

    root->left->right = NULL;

    root->right = (struct *node* \*)malloc(sizeof(struct *node*));

    root->right->data = 3;

    root->right->left = NULL;

    root->right->right = NULL;

    printf("Inorder:-");

    inorderDfs(root);

    printf("Preorder:-");

    preorderDfs(root);

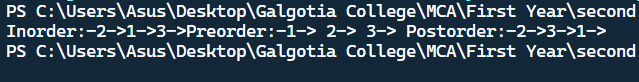
    printf("Postorder:-");

    postorderDfs(root);

    return 0;

}

**Output:-**

****

**Program:-10**

**Object:-** Write a program to implement linear search.

**Code:-**

#include <stdio.h>

*// Function to perform linear search*

int linearSearch(int arr*[]*, int n, int target) {

    for (int i = 0; i < n; i++) {

        if (arr[i] == target) {

            return i; *// Return the index of the found element*

        }

    }

    return -1; *// Return -1 if the element is not found*

}

int main() {

    int arr*[]* = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};

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

    int target;

    printf("Enter the element to search: ");

    scanf("%d", &target);

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

    if (result != -1) {

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

    } else {

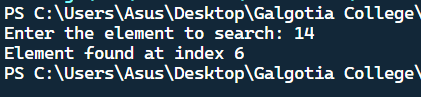
        printf("Element not found in the array\n");

    }

    return 0;

}

**Output:-**

****

**Program:-11**

**Object:-**Write a program to implement binary search.

**Code:-**

#include <stdio.h>

*// Function to perform binary search*

int binarySearch(int arr*[]*, int low, int high, int target) {

    while (low <= high) {

        int mid = low + (high - low) / 2;

*// Check if target is present at mid*

        if (arr[mid] == target) {

            return mid;

        }

*// If target greater, ignore the left half*

        if (arr[mid] < target) {

            low = mid + 1;

        }

*// If target is smaller, ignore the right half*

        else {

            high = mid - 1;

        }

    }

*// If the element is not present in the array*

    return -1;

}

int main() {

    int arr*[]* = {2, 3, 4, 10, 40};

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

    int target;

    printf("Enter the element to search: ");

    scanf("%d", &target);

    int result = binarySearch(arr, 0, n - 1, target);

    if (result != -1) {

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

    } else {

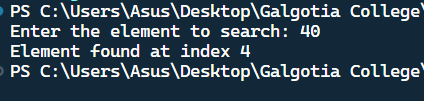
        printf("Element not found in the array\n");

    }

    return 0;

}

**Output:-**

****

**Program:-12**

**Object:-**Write a program to implement the bubble sort.

**Code:-**

#include <stdio.h>

*// Function to perform bubble sort*

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

{

    for (int i = 0; i < n - 1; i++)

    {

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

        {

            if (arr[j] > arr[j + 1])

            {

*// Swap arr[j] and arr[j+1]*

                int temp = arr[j];

                arr[j] = arr[j + 1];

                arr[j + 1] = temp;

            }

        }

    }

}

*// Function to print an array*

void printArray(int arr*[]*, int size)

{

    for (int i = 0; i < size; i++)

    {

        printf("%d ", arr[i]);

    }

    printf("\n");

}

int main()

{

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

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

    printf("Unsorted array: ");

    printArray(arr, n);

    bubbleSort(arr, n);

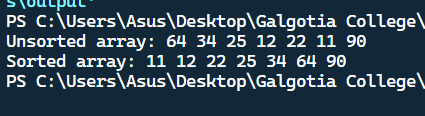
    printf("Sorted array: ");

    printArray(arr, n);

    return 0;

}

**Output:-**

****

**Program:-13**

**Object:-**Write a program to implement selection sort.

**Code:-**

#include <stdio.h>

*// Function to perform selection sort*

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

{

    for (int i = 0; i < n - 1; i++)

    {

*// Find the minimum element in the unsorted part of the array*

        int minIdx = i;

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

        {

            if (arr[j] < arr[minIdx])

            {

                minIdx = j;

            }

        }

*// Swap the found minimum element with the first element of the unsorted part*

        int temp = arr[minIdx];

        arr[minIdx] = arr[i];

        arr[i] = temp;

    }

}

*// Function to print an array*

void printArray(int arr*[]*, int size)

{

    for (int i = 0; i < size; i++)

    {

        printf("%d ", arr[i]);

    }

    printf("\n");

}

int main()

{

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

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

    printf("Unsorted array: ");

    printArray(arr, n);

    selectionSort(arr, n);

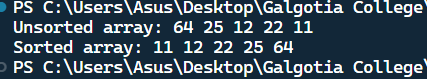
    printf("Sorted array: ");

    printArray(arr, n);

    return 0;

}

**Output:-**



**Program:-14**

**Object:-**Write a program to implement a insertion sort.

**Code:-**

#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];

        int j = i - 1;

*// Move elements of arr[0..i-1] that are greater than key*

*// to one position ahead of their current position*

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

        {

            arr[j + 1] = arr[j];

            j = j - 1;

        }

        arr[j + 1] = key;

    }

}

*// Function to print an array*

void printArray(int arr*[]*, int size)

{

    for (int i = 0; i < size; i++)

    {

        printf("%d ", arr[i]);

    }

    printf("\n");

}

int main()

{

    int arr*[]* = {12, 11, 13, 5, 6};

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

    printf("Unsorted array: ");

    printArray(arr, n);

    insertionSort(arr, n);

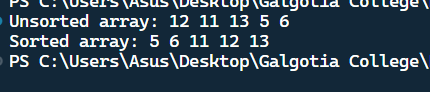
    printf("Sorted array: ");

    printArray(arr, n);

    return 0;

}

**Output:-**

****

**Program:-15**

**Object:-**Write a program to implement merge sort.

**Code:-**

#include <stdio.h>

#include <stdlib.h>

*// Function to merge two subarrays of arr[]*

void merge(int arr*[]*, int l, int m, int r) {

    int i, j, k;

    int n1 = m - l + 1;

    int n2 = r - m;

*// Create temporary arrays*

    int L[n1], R[n2];

*// Copy data to temporary arrays L[] and R[]*

    for (i = 0; i < n1; i++)

        L[i] = arr[l + i];

    for (j = 0; j < n2; j++)

        R[j] = arr[m + 1 + j];

*// Merge the temporary arrays back into arr[l..r]*

    i = 0; *// Initial index of first subarray*

    j = 0; *// Initial index of second subarray*

    k = l; *// Initial index of merged subarray*

    while (i < n1 && j < n2) {

        if (L[i] <= R[j]) {

            arr[k] = L[i];

            i++;

        } else {

            arr[k] = R[j];

            j++;

        }

        k++;

    }

*// Copy the remaining elements of L[], if there are any*

    while (i < n1) {

        arr[k] = L[i];

        i++;

        k++;

    }

*// Copy the remaining elements of R[], if there are any*

    while (j < n2) {

        arr[k] = R[j];

        j++;

        k++;

    }

}

*// Function to implement merge sort*

void mergeSort(int arr*[]*, int l, int r) {

    if (l < r) {

*// Find the middle point*

        int m = l + (r - l) / 2;

*// Sort first and second halves*

        mergeSort(arr, l, m);

        mergeSort(arr, m + 1, r);

*// Merge the sorted halves*

        merge(arr, l, m, r);

    }

}

*// Function to print an array*

void printArray(int arr*[]*, int size) {

    for (int i = 0; i < size; i++) {

        printf("%d ", arr[i]);

    }

    printf("\n");

}

int main() {

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

    int arr\_size = sizeof(arr) / sizeof(arr[0]);

    printf("Unsorted array: ");

    printArray(arr, arr\_size);

    mergeSort(arr, 0, arr\_size - 1);

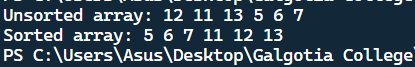
    printf("Sorted array: ");

    printArray(arr, arr\_size);

    return 0;

}

**Output:-**

****

**Program:-16**

**Object:-**Write a program to implement heap sort.

**Code:-**

#include <stdio.h>

*// Function to heapify a subtree rooted at node i which is an index in arr[]*

*// n is the size of the heap*

void heapify(int arr*[]*, int n, int i) {

    int largest = i; *// Initialize largest as root*

    int left = 2 \* i + 1; *// left = 2\*i + 1*

    int right = 2 \* i + 2; *// right = 2\*i + 2*

*// If left child is larger than root*

    if (left < n && arr[left] > arr[largest])

        largest = left;

*// If right child is larger than largest so far*

    if (right < n && arr[right] > arr[largest])

        largest = right;

*// If largest is not root*

    if (largest != i) {

        int swap = arr[i];

        arr[i] = arr[largest];

        arr[largest] = swap;

*// Recursively heapify the affected sub-tree*

        heapify(arr, n, largest);

    }

}

*// Function to perform heap sort*

void heapSort(int arr*[]*, int n) {

*// Build heap (rearrange array)*

    for (int i = n / 2 - 1; i >= 0; i--)

        heapify(arr, n, i);

*// One by one extract an element from heap*

    for (int i = n - 1; i > 0; i--) {

*// Move current root to end*

        int temp = arr[0];

        arr[0] = arr[i];

        arr[i] = temp;

*// Call max heapify on the reduced heap*

        heapify(arr, i, 0);

    }

}

*// Function to print an array*

void printArray(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("Unsorted array: ");

    printArray(arr, n);

    heapSort(arr, n);

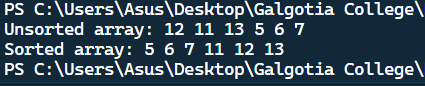
    printf("Sorted array: ");

    printArray(arr, n);

    return 0;

}

**Output:-**

****

**Program:-17**

**Object:-**Write a program to implement matrix multiplication using strassen’s algorithm.

**Code:-**

#include<stdio.h>

int main(){

  int a[2][2], b[2][2], c[2][2], i, j;

  int m1, m2, m3, m4 , m5, m6, m7;

  printf("Enter the 4 elements of first matrix: ");

  for(i = 0;i < 2; i++)

      for(j = 0;j < 2; j++)

           scanf("%d", &a[i][j]);

  printf("Enter the 4 elements of second matrix: ");

  for(i = 0; i < 2; i++)

      for(j = 0;j < 2; j++)

           scanf("%d", &b[i][j]);

  printf("\nThe first matrix is\n");

  for(i = 0; i < 2; i++){

      printf("\n");

      for(j = 0; j < 2; j++)

           printf("%d\t", a[i][j]);

  }

  printf("\nThe second matrix is\n");

  for(i = 0;i < 2; i++){

      printf("\n");

      for(j = 0;j < 2; j++)

           printf("%d\t", b[i][j]);

  }

  m1= (a[0][0] + a[1][1]) \* (b[0][0] + b[1][1]);

  m2= (a[1][0] + a[1][1]) \* b[0][0];

  m3= a[0][0] \* (b[0][1] - b[1][1]);

  m4= a[1][1] \* (b[1][0] - b[0][0]);

  m5= (a[0][0] + a[0][1]) \* b[1][1];

  m6= (a[1][0] - a[0][0]) \* (b[0][0]+b[0][1]);

  m7= (a[0][1] - a[1][1]) \* (b[1][0]+b[1][1]);

  c[0][0] = m1 + m4- m5 + m7;

  c[0][1] = m3 + m5;

  c[1][0] = m2 + m4;

  c[1][1] = m1 - m2 + m3 + m6;

   printf("\nAfter multiplication using Strassen's algorithm \n");

   for(i = 0; i < 2 ; i++){

      printf("\n");

      for(j = 0;j < 2; j++)

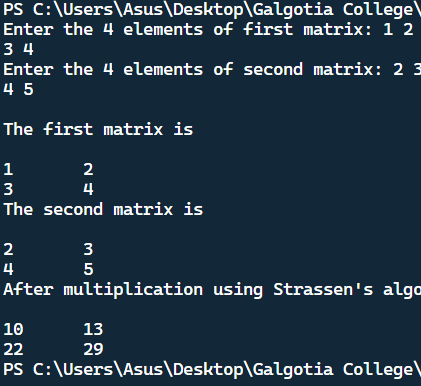
           printf("%d\t", c[i][j]);

   }

   return 0;

}

**Output:-**

****

**Program:-18**

**Object:-**Write a program to find minimum spanning tree using kruskal’s algorithm.

**Code:-**

#include <stdio.h>

#include <stdlib.h>

*// Comparator function to use in sorting*

int comparator(*const* void \*p1, *const* void \*p2)

{

*const* int(\*x)[3] = p1;

*const* int(\*y)[3] = p2;

    return (\*x)[2] - (\*y)[2];

}

*// Initialization of parent[] and rank[] arrays*

void makeSet(int parent*[]*, int rank*[]*, int n)

{

    for (int i = 0; i < n; i++)

    {

        parent[i] = i;

        rank[i] = 0;

    }

}

*// Function to find the parent of a node*

int findParent(int parent*[]*, int component)

{

    if (parent[component] == component)

        return component;

    return parent[component] = findParent(parent, parent[component]);

}

*// Function to unite two sets*

void unionSet(int u, int v, int parent*[]*, int rank*[]*, int n)

{

*// Finding the parents*

    u = findParent(parent, u);

    v = findParent(parent, v);

    if (rank[u] < rank[v])

    {

        parent[u] = v;

    }

    else if (rank[u] > rank[v])

    {

        parent[v] = u;

    }

    else

    {

        parent[v] = u;

*// Since the rank increases if*

*// the ranks of two sets are same*

        rank[u]++;

    }

}

*// Function to find the MST*

void kruskalAlgo(int n, int edge[n][3])

{

*// First we sort the edge array in ascending order*

*// so that we can access minimum distances/cost*

    qsort(edge, n, sizeof(edge[0]), comparator);

    int parent[n];

    int rank[n];

*// Function to initialize parent[] and rank[]*

    makeSet(parent, rank, n);

*// To store the minimun cost*

    int minCost = 0;

    printf(

        "Following are the edges in the constructed MST\n");

    for (int i = 0; i < n; i++)

    {

        int v1 = findParent(parent, edge[i][0]);

        int v2 = findParent(parent, edge[i][1]);

        int wt = edge[i][2];

*// If the parents are different that*

*// means they are in different sets so*

*// union them*

        if (v1 != v2)

        {

            unionSet(v1, v2, parent, rank, n);

            minCost += wt;

            printf("%d -- %d == %d\n", edge[i][0],

                   edge[i][1], wt);

        }

    }

    printf("Minimum Cost Spanning Tree: %d\n", minCost);

}

*// Driver code*

int main()

{

    int edge[5][3] = {{0, 1, 10},

                      {0, 2, 6},

                      {0, 3, 5},

                      {1, 3, 15},

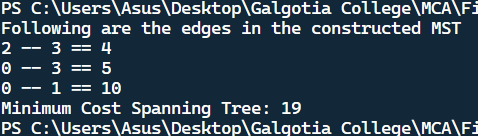
                      {2, 3, 4}};

    kruskalAlgo(5, edge);

    return 0;

}

**Output:-**

****