**1. Write a C program to perform Matrix Multiplication**

#include <stdio.h>

#define MAX 10

void multiplyMatrices(int first[MAX][MAX], int second[MAX][MAX], int result[MAX][MAX], int rowFirst, int columnFirst, int rowSecond, int columnSecond) {

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

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

result[i][j] = 0;

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

result[i][j] += first[i][k] \* second[k][j];

}

}

}

}

void displayMatrix(int matrix[MAX][MAX], int row, int column) {

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

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

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

}

printf("\n");

}

}

int main() {

int first[MAX][MAX], second[MAX][MAX], result[MAX][MAX], rowFirst, columnFirst, rowSecond, columnSecond;

printf("Enter rows and columns for first matrix: ");

scanf("%d %d", &rowFirst, &columnFirst);

printf("Enter rows and columns for second matrix: ");

scanf("%d %d", &rowSecond, &columnSecond);

if (columnFirst != rowSecond) {

printf("Error! column of first matrix not equal to row of second.\n");

return 0;

}

printf("Enter elements of matrix 1:\n");

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

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

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

}

}

printf("Enter elements of matrix 2:\n");

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

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

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

}

}

multiplyMatrices(first, second, result, rowFirst, columnFirst, rowSecond, columnSecond);

printf("Resultant Matrix:\n");

displayMatrix(result, rowFirst, columnSecond);

return 0;

}

**2. Write a C program to find Odd or Even number from a given set of numbers**

#include <stdio.h>

int main() {

int n;

// Ask user for the number of elements in the set

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

scanf("%d", &n);

int numbers[n];

// Input the numbers

printf("Enter the numbers:\n");

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

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

}

// Check whether each number is odd or even

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

if (numbers[i] % 2 == 0) {

printf("%d is Even\n", numbers[i]);

} else {

printf("%d is Odd\n", numbers[i]);

}

}

return 0;

}

**3. Write a C program to find Factorial of a given number without using Recursion**

#include <stdio.h>

int main() {

int num;

unsigned long long factorial = 1; // Using unsigned long long to handle large results

// Ask the user to enter a number

printf("Enter a positive integer: ");

scanf("%d", &num);

// Check if the number is negative, since factorial is not defined for negative numbers

if (num < 0) {

printf("Factorial is not defined for negative numbers.\n");

} else {

// Calculate the factorial using a loop

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

factorial \*= i;

}

// Output the result

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

}

return 0;

}

**4. Given a number n. the task is to print the Fibonacci series and the sum**

**of the series using recursion.**

**input: n=10**

**output: Fibonacci series**

**0, 1, 1, 2, 3, 5, 8, 13, 21, 34**

**Sum: 88**

#include <stdio.h>

int fibonacci(int n)

{

if (n <= 1)

return n;

return fibonacci(n -1)+fibonacci(n -2);

}

void printFibonacci(int n)

{

int sum = 0;

printf("Fibonacci series: ");

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

{

int fib = fibonacci(i);

sum += fib;

printf("%d, ", fib);

}

printf("\nSum: %d\n", sum);

}

int main()

{

int n = 10;

printFibonacci(n);

return 0;

}

**5. Write a C program to find Factorial of a given number using Recursion**

#include <stdio.h>

// Function to calculate factorial using recursion

unsigned long long factorial(int n) {

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

return 1; // Base case: factorial of 0 and 1 is 1

} else {

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

}

}

int main() {

int num;

// Ask the user to enter a number

printf("Enter a positive integer: ");

scanf("%d", &num);

// Check if the number is negative, since factorial is not defined for negative numbers

if (num < 0) {

printf("Factorial is not defined for negative numbers.\n");

} else {

// Call the factorial function and display the result

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

}

return 0;

}

**6. Write a C program to find Fibonacci series using Recursion**

#include <stdio.h>

// Function to calculate Fibonacci number at position n

int fibonacci(int n) {

if (n <= 1) {

return n; // Base case: Fibonacci of 0 is 0 and Fibonacci of 1 is 1

} else {

return fibonacci(n - 1) + fibonacci(n - 2); // Recursive case

}

}

int main() {

int num;

// Ask the user for the number of terms in the Fibonacci series

printf("Enter the number of terms for Fibonacci series: ");

scanf("%d", &num);

printf("Fibonacci Series up to %d terms: \n", num);

// Display Fibonacci series up to the entered number of terms

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

printf("%d ", fibonacci(i));

}

printf("\n");

return 0;

}

**7. Write a C program to implement Array operations such as Insert, Delete and**

**Display**

#include <stdio.h>

#define MAX 100

void insert(int arr[], int \*n, int value, int position) {

for (int i = \*n; i > position; i--) {

arr[i] = arr[i - 1];

}

arr[position] = value;

(\*n)++;

}

void delete(int arr[], int \*n, int position) {

for (int i = position; i < \*n - 1; i++) {

arr[i] = arr[i + 1];

}

(\*n)--;

}

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

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

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

}

printf("\n");

}

int main() {

int arr[MAX], n = 0, choice, value, position;

while (1) {

printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to insert: ");

scanf("%d", &value);

printf("Enter position to insert: ");

scanf("%d", &position);

insert(arr, &n, value, position);

break;

case 2:

printf("Enter position to delete: ");

scanf("%d", &position);

delete(arr, &n, position);

break;

case 3:

display(arr, n);

break;

case 4:

return 0;

default:

printf("Invalid choice!\n");

}

}

return 0;

}

**8. Write a C program to search a number using Linear Search method**

#include <stdio.h>

int linearSearch(int arr[], int size, int target) {

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

if (arr[i] == target) {

return i; // Return the index of the target

}

}

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

}

int main() {

int arr[] = {5, 3, 8, 4, 2};

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

int target = 4;

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

if (result != -1) {

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

} else {

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

}

return 0;

}

**9. You are given an array arr in increasing order. Find the element x from**

**arr using binary search.**

**Example 1: arr={ 1,5,6,7,9,10},X=6**

**Output : Element found at location 2**

**Example 2: arr={ 1,5,6,7,9,10},X=11**

**Output : Element not found at location 2**

#include <stdio.h>

int binarySearch(int arr[], int left, int right, int x) {

while (left <= right) {

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

if (arr[mid] == x)

return mid;

if (arr[mid] < x)

left = mid + 1;

else

right = mid -1;

}

return -1;

}

int main() {

int arr[] = {1, 5, 6, 7, 9, 10};

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

int x;

scanf("%d", &x);

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

if (result != -1)

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

else

printf("Element %d not found in the array.\n", x);

return 0;

}

**10.Write a C program to implement Linked list operations**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

void insertAtBeginning(struct Node\*\* head\_ref, int new\_data) {

struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));

new\_node->data = new\_data;

new\_node->next = (\*head\_ref);

(\*head\_ref) = new\_node;

}

void displayList(struct Node\* node) {

while (node != NULL) {

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

node = node->next;

}

printf("NULL\n");

}

int main() {

struct Node\* head = NULL;

insertAtBeginning(&head, 3);

insertAtBeginning(&head, 7);

insertAtBeginning(&head, 9);

printf("Linked list: ");

displayList(head);

return 0;

}

**11. Write a C program to implement Stack operations such as PUSH,**

**POP and PEEK**

#include <stdio.h>

#include <stdlib.h>

#define MAX 10

struct Stack {

int top;

int items[MAX];

};

struct Stack\* createStack() {

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

stack->top = -1;

return stack;

}

int isFull(struct Stack\* stack) {

return stack->top == MAX - 1;

}

int isEmpty(struct Stack\* stack) {

return stack->top == -1;

}

void push(struct Stack\* stack, int item) {

if (isFull(stack)) {

printf("Stack Overflow\n");

} else {

stack->items[++stack->top] = item;

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

}

}

int pop(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack Underflow\n");

return -1;

} else {

return stack->items[stack->top--];

}

}

int peek(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack is empty\n");

return -1;

} else {

return stack->items[stack->top];

}

}

int main() {

struct Stack\* stack = createStack();

push(stack, 10);

push(stack, 20);

push(stack, 30);

printf("%d popped from stack\n", pop(stack));

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

return 0;

}

**12. Write a C program to implement the application of Stack (Notations)**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX 100

typedef struct Stack {

int top;

char items[MAX];

} Stack;

void initStack(Stack\* s) {

s->top = -1;

}

int isFull(Stack\* s) {

return s->top == MAX - 1;

}

int isEmpty(Stack\* s) {

return s->top == -1;

}

void push(Stack\* s, char item) {

if (!isFull(s)) {

s->items[++(s->top)] = item;

}

}

char pop(Stack\* s) {

if (!isEmpty(s)) {

return s->items[(s->top)--];

}

return '\0';

}

char peek(Stack\* s) {

if (!isEmpty(s)) {

return s->items[s->top];

}

return '\0';

}

int precedence(char op) {

if (op == '+' || op == '-') return 1;

if (op == '\*' || op == '/') return 2;

return 0;

}

void infixToPostfix(char\* infix, char\* postfix) {

Stack s;

initStack(&s);

int j = 0;

for (int i = 0; i < strlen(infix); i++) {

if (isalnum(infix[i])) {

postfix[j++] = infix[i];

} else if (infix[i] == '(') {

push(&s, infix[i]);

} else if (infix[i] == ')') {

while (!isEmpty(&s) && peek(&s) != '(') {

postfix[j++] = pop(&s);

}

pop(&s);

} else {

while (!isEmpty(&s) && precedence(peek(&s)) >= precedence(infix[i])) {

postfix[j++] = pop(&s);

}

push(&s, infix[i]);

}

}

while (!isEmpty(&s)) {

postfix[j++] = pop(&s);

}

postfix[j] = '\0';

}

int main() {

char infix[MAX], postfix[MAX];

printf("Enter an infix expression: ");

scanf("%s", infix);

infixToPostfix(infix, postfix);

printf("Postfix expression: %s\n", postfix);

return 0;

}

**13. Write a C program to implement Queue operations such as ENQUEUE, DEQUEUE**

**and Display**

#include <stdio.h>

#include <stdlib.h>

#define MAX 5

struct Queue {

int items[MAX];

int front;

int rear;

};

struct Queue\* createQueue() {

struct Queue\* q = (struct Queue\*)malloc(sizeof(struct Queue));

q->front = -1;

q->rear = -1;

return q;

}

int isFull(struct Queue\* q) {

return q->rear == MAX - 1;

}

int isEmpty(struct Queue\* q) {

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

}

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

if (isFull(q)) {

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

} else {

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

q->front = 0;

}

q->rear++;

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

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

}

}

int dequeue(struct Queue\* q) {

if (isEmpty(q)) {

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

return -1;

} else {

int item = q->items[q->front];

q->front++;

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

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

}

return item;

}

}

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->items[i]);

}

printf("\n");

}

}

int main() {

struct Queue\* q = createQueue();

enqueue(q, 10);

enqueue(q, 20);

enqueue(q, 30);

display(q);

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

display(q);

enqueue(q, 40);

enqueue(q, 50);

enqueue(q, 60);

display(q);

return 0;

}

**14. .Given two integer arrays preorder and inorder where preorder is the**

**preorder**

**traversal of a binary tree and inorder is the inorder traversal of the same**

**tree,**

**construct and return the binary tree.**

**Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]**

**Output: [3,9,20,null,null,15,7]**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

int val;

struct TreeNode \*left;

struct TreeNode \*right;

};

struct TreeNode\* buildTree(int\* preorder, int\* inorder, int inStart, int

inEnd, int\* preIndex) {

if (inStart > inEnd)

return NULL;

struct TreeNode\* root = (struct TreeNode\*)malloc(sizeof(struct

TreeNode));

root->val = preorder[(\*preIndex)++];

int inIndex;

for (inIndex = inStart; inIndex <= inEnd; inIndex++) {

if (inorder[inIndex] == root->val)

break;

}

root->left = buildTree(preorder, inorder, inStart, inIndex -1, preIndex);

root->right = buildTree(preorder, inorder, inIndex + 1, inEnd, preIndex);

return root;

}

void printLevelOrder(struct TreeNode\* root) {

if (root == NULL)

return;

struct TreeNode\* queue[100];

int front = -1, rear = -1;

queue[++rear] = root;

while (front != rear) {

struct TreeNode\* temp = queue[++front];

if (temp != NULL) {

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

if (temp->left || temp->right) {

queue[++rear] = temp->left;

queue[++rear] = temp->right;

}

if (front != rear)

printf(",");

} else {

printf("null");

if (front != rear)

printf(",");

}

}

}

int main() {

int preorder[] = {3, 9, 20, 15, 7};

int inorder[] = {9, 3, 15, 20, 7};

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

int preIndex = 0;

struct TreeNode\* root = buildTree(preorder, inorder, 0, n -1, &preIndex);

printf("[");

printLevelOrder(root);

printf("]\n");

return 0;

}

**15. Write a C program to implement hashing using Linear Probing method**

#include <stdio.h>

#include <stdlib.h>

#define TABLE\_SIZE 10

typedef struct {

int key;

int value;

} HashEntry;

HashEntry\* hashTable[TABLE\_SIZE];

int hashFunction(int key) {

return key % TABLE\_SIZE;

}

void insert(int key, int value) {

int index = hashFunction(key);

while (hashTable[index] != NULL) {

index = (index + 1) % TABLE\_SIZE;

}

hashTable[index] = (HashEntry\*)malloc(sizeof(HashEntry));

hashTable[index]->key = key;

hashTable[index]->value = value;

}

HashEntry\* search(int key) {

int index = hashFunction(key);

while (hashTable[index] != NULL) {

if (hashTable[index]->key == key) {

return hashTable[index];

}

index = (index + 1) % TABLE\_SIZE;

}

return NULL;

}

void display() {

for (int i = 0; i < TABLE\_SIZE; i++) {

if (hashTable[i] != NULL) {

printf("Index %d: Key = %d, Value = %d\n", i, hashTable[i]->key, hashTable[i]->value);

} else {

printf("Index %d: Empty\n", i);

}

}

}

int main() {

insert(1, 10);

insert(2, 20);

insert(12, 30);

insert(22, 40);

display();

HashEntry\* entry = search(12);

if (entry != NULL) {

printf("Found: Key = %d, Value = %d\n", entry->key, entry->value);

} else {

printf("Key not found.\n");

}

return 0;

}

**16. You are given with an array arr which contains integer elements. Sort**

**these elements in ascending order using insertion sort and print the 6th**

**Iteration result.**

**Example:**

**Input:98,23,45,14,6,67,33,42**

**Output:6,14,23,33,45,67,98,42**

#include <stdio.h>

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

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

int key = arr[i];

int j = i -1;

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

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

j--;

}

arr[j + 1] = key;

}

}

int main()

{

int arr[] = {98, 23, 45, 14, 6, 67, 33, 42};

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

insertionSort(arr, n);

printf("Result after 6th iteration: ");

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

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

if (i < n -1) {

printf(",");

}

}

printf("\n");

return 0;

}

**17.Write a C program to arrange a series of numbers using Merge Sort**

#include <stdio.h>

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

int i, j, k;

int n1 = mid - left + 1;

int n2 = right - mid;

int L[n1], R[n2];

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

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

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

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

i = 0;

j = 0;

k = left;

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

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

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

if (left < right) {

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

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

int main() {

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

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

mergeSort(arr, 0, arr\_size - 1);

printf("Sorted array: \n");

for (int i = 0; i < arr\_size; i++)

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

printf("\n");

return 0;

}

**18.Write a C program to arrange a series of numbers using Quick Sort**

#include <stdio.h>

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

int t = \*a;

\*a = \*b;

\*b = t;

}

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

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j <= high - 1; j++) {

if (arr[j] < pivot) {

i++;

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

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

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

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

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

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

}

printf("\n");

}

int main() {

int arr[] = {10, 7, 8, 9, 1, 5};

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

printf("Unsorted array: \n");

printArray(arr, n);

quickSort(arr, 0, n - 1);

printf("Sorted array: \n");

printArray(arr, n);

return 0;

}

**19.Write a C program to implement Heap sort**

#include <stdio.h>

// Function to swap two elements

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

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to heapify a subtree rooted at index i

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

int largest = i; // Initialize largest as root

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

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

// 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) {

swap(&arr[i], &arr[largest]);

// Recursively heapify the affected sub-tree

heapify(arr, n, largest);

}

}

// Main 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

swap(&arr[0], &arr[i]);

// 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");

}

// Driver code

int main() {

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

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

printf("Original array: \n");

printArray(arr, n);

heapSort(arr, n);

printf("Sorted array: \n");

printArray(arr, n);

return 0;

}

**20.Write a program to perform the following operations:**

**a) Insert an element into a AVL tree**

**b) Delete an element from a AVL tree**

**c) Search for a key element in a AVL tree**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int key;

struct Node \*left;

struct Node \*right;

int height;

};

int max(int a, int b) {

return (a > b) ? a : b;

}

int height(struct Node \*N) {

if (N == NULL)

return 0;

return N->height;

}

struct Node\* newNode(int key) {

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

node->key = key;

node->left = NULL;

node->right = NULL;

node->height = 1;

return node;

}

struct Node \*rightRotate(struct Node \*y) {

struct Node \*x = y->left;

struct Node \*T2 = x->right;

x->right = y;

y->left = T2;

y->height = max(height(y->left), height(y->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1;

return x;

}

struct Node \*leftRotate(struct Node \*x) {

struct Node \*y = x->right;

struct Node \*T2 = y->left;

y->left = x;

x->right = T2;

x->height = max(height(x->left), height(x->right)) + 1;

y->height = max(height(y->left), height(y->right)) + 1;

return y;

}

int getBalance(struct Node \*N) {

if (N == NULL)

return 0;

return height(N->left) - height(N->right);

}

struct Node\* insert(struct Node\* node, int key) {

if (node == NULL)

return newNode(key);

if (key < node->key)

node->left = insert(node->left, key);

else if (key > node->key)

node->right = insert(node->right, key);

else

return node;

node->height = 1 + max(height(node->left), height(node->right)));

int balance = getBalance(node);

if (balance > 1 && key < node->left->key)

return rightRotate(node);

if (balance < -1 && key > node->right->key)

return leftRotate(node);

if (balance > 1 && key > node->left->key) {

node->left = leftRotate(node->left);

return rightRotate(node);

}

if (balance < -1 && key < node->right->key) {

node->right = rightRotate(node->right);

return leftRotate(node);

}

return node;

}

struct Node\* minValueNode(struct Node\* node) {

struct Node\* current = node;

while (current->left != NULL)

current = current->left;

return current;

}

struct Node\* deleteNode(struct Node\* root, int key) {

if (root == NULL)

return root;

if (key < root->key)

root->left = deleteNode(root->left, key);

else if (key > root->key)

root->right = deleteNode(root->right, key);

else {

if ((root->left == NULL) || (root->right == NULL)) {

struct Node \*temp = root->left ? root->left : root->right;

if (temp == NULL) {

temp = root;

root = NULL;

} else

\*root = \*temp;

free(temp);

} else {

struct Node\* temp = minValueNode(root->right);

root->key = temp->key;

root->right = deleteNode(root->right, temp->key);

}

}

if (root == NULL)

return root;

root->height = 1 + max(height(root->left), height(root->right)));

int balance = getBalance(root);

if (balance > 1 && getBalance(root->left) >= 0)

return rightRotate(root);

if (balance > 1 && getBalance(root->left) < 0) {

root->left = leftRotate(root->left);

return rightRotate(root);

}

if (balance < -1 && getBalance(root->right) <= 0)

return leftRotate(root);

if (balance < -1 && getBalance(root->right) > 0) {

root->right = rightRotate(root->right);

return leftRotate(root);

}

return root;

}

struct Node\* search(struct Node\* root, int key) {

if (root == NULL || root->key == key)

return root;

if (root->key < key)

return search(root->right, key);

return search(root->left, key);

}

void preOrder(struct Node \*root) {

if (root != NULL) {

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

preOrder(root->left);

preOrder(root->right);

}

}

int main() {

struct Node \*root = NULL;

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 30);

root = insert(root, 40);

root = insert(root, 50);

root = insert(root, 25);

printf("Preorder traversal of the AVL tree is \n");

preOrder(root);

root = deleteNode(root, 10);

printf("\nPreorder traversal after deletion of 10 \n");

preOrder(root);

struct Node\* foundNode = search(root, 30);

if (foundNode != NULL)

printf("\nElement 30 found in the AVL tree.\n");

else

printf("\nElement 30 not found in the AVL tree.\n");

return 0;

}

**21. Write a C program to Graph traversal using Breadth First Search**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX 100

int graph[MAX][MAX], visited[MAX], queue[MAX], front = -1, rear = -1;

int n;

void enqueue(int vertex) {

if (rear == MAX - 1) {

printf("Queue is full\n");

} else {

if (front == -1) {

front = 0;

}

rear++;

queue[rear] = vertex;

}

}

int dequeue() {

int vertex;

if (front == -1) {

printf("Queue is empty\n");

return -1;

} else {

vertex = queue[front];

front++;

if (front > rear) {

front = rear = -1;

}

return vertex;

}

}

bool isEmpty() {

return front == -1;

}

void bfs(int startVertex) {

visited[startVertex] = 1;

enqueue(startVertex);

while (!isEmpty()) {

int currentVertex = dequeue();

printf("%d ", currentVertex);

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

if (graph[currentVertex][i] == 1 && !visited[i]) {

visited[i] = 1;

enqueue(i);

}

}

}

}

int main() {

int edges, startVertex;

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

scanf("%d", &n);

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

scanf("%d", &edges);

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

int u, v;

printf("Enter edge (u v): ");

scanf("%d %d", &u, &v);

graph[u][v] = 1;

graph[v][u] = 1; // For undirected graph

}

printf("Enter the starting vertex for BFS: ");

scanf("%d", &startVertex);

printf("Breadth First Search starting from vertex %d: ", startVertex);

bfs(startVertex);

return 0;

}

**22. Write a C program to Graph traversal using Depth First Search**

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

int visited[MAX];

int adjMatrix[MAX][MAX];

int n;

void depthFirstSearch(int vertex) {

int i;

visited[vertex] = 1;

printf("%d ", vertex);

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

if (adjMatrix[vertex][i] && !visited[i]) {

depthFirstSearch(i);

}

}

}

int main() {

int i, j, startVertex;

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

scanf("%d", &n);

printf("Enter the adjacency matrix:\n");

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

for (j = 0; j < n; j++) {

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

}

}

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

visited[i] = 0;

}

printf("Enter the starting vertex for DFS: ");

scanf("%d", &startVertex);

printf("Depth First Search starting from vertex %d: ", startVertex);

depthFirstSearch(startVertex);

return 0;

}

**23. Implementation of Shortest Path Algorithms using Dijkstra’s Algorithm**

// Implementation of Dijkstra's Algorithm for finding the shortest path in a graph

#include <stdio.h>

#include <limits.h>

#define V 9

int minDistance(int dist[], bool sptSet[]) {

int min = INT\_MAX, min\_index;

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

if (sptSet[v] == false && dist[v] <= min)

min = dist[v], min\_index = v;

return min\_index;

}

void printSolution(int dist[]) {

printf("Vertex \t Distance from Source\n");

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

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

}

void dijkstra(int graph[V][V], int src) {

int dist[V];

bool sptSet[V];

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

dist[i] = INT\_MAX, sptSet[i] = false;

dist[src] = 0;

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

int u = minDistance(dist, sptSet);

sptSet[u] = true;

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

if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX

&& dist[u]+graph[u][v] < dist[v])

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

}

printSolution(dist);

}

int main() {

int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},

{4, 0, 8, 0, 0, 0, 0, 11, 0},

{0, 8, 0, 7, 0, 4, 0, 0, 2},

{0, 0, 7, 0, 9, 14, 0, 0, 0},

{0, 0, 0, 9, 0, 10, 0, 0, 0},

{0, 0, 4, 14, 10, 0, 2, 0, 0},

{0, 0, 0, 0, 0, 2, 0, 1, 6},

{8, 11, 0, 0, 0, 0, 1, 0, 7},

{0, 0, 2, 0, 0, 0, 6, 7, 0}};

dijkstra(graph, 0);

return 0;

}

**24.Implementation of Minimum Spanning Tree using Prim’s Algorithm**

#include <stdio.h>

#include <limits.h>

#define V 5

int minKey(int key[], int mstSet[]) {

int min = INT\_MAX, min\_index;

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

if (mstSet[v] == 0 && key[v] < min) {

min = key[v];

min\_index = v;

}

}

return min\_index;

}

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

int parent[V];

int key[V];

int mstSet[V];

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

key[i] = INT\_MAX;

mstSet[i] = 0;

}

key[0] = 0;

parent[0] = -1;

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

int u = minKey(key, mstSet);

mstSet[u] = 1;

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

if (graph[u][v] && mstSet[v] == 0 && 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 graph[V][V] = { { 0, 2, 0, 6, 0 },

{ 2, 0, 3, 8, 5 },

{ 0, 3, 0, 0, 7 },

{ 6, 8, 0, 0, 9 },

{ 0, 5, 7, 9, 0 } };

primMST(graph);

return 0;

}

**25.Implementation of Minimum Spanning Tree using Kruskal Algorithm**

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

#define INF 99999

typedef struct {

int u, v, weight;

} Edge;

Edge edges[MAX];

int parent[MAX];

int rank[MAX];

int numEdges, numVertices;

void initialize() {

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

parent[i] = i;

rank[i] = 0;

}

}

int find(int i) {

if (parent[i] != i) {

parent[i] = find(parent[i]);

}

return parent[i];

}

void unionSets(int u, int v) {

int rootU = find(u);

int rootV = find(v);

if (rootU != rootV) {

if (rank[rootU] > rank[rootV]) {

parent[rootV] = rootU;

} else if (rank[rootU] < rank[rootV]) {

parent[rootU] = rootV;

} else {

parent[rootV] = rootU;

rank[rootU]++;

}

}

}

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

return ((Edge \*)a)->weight - ((Edge \*)b)->weight;

}

void kruskal() {

int mstWeight = 0;

initialize();

qsort(edges, numEdges, sizeof(edges[0]), compareEdges);

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

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

int u = edges[i].u;

int v = edges[i].v;

int weight = edges[i].weight;

if (find(u) != find(v)) {

unionSets(u, v);

mstWeight += weight;

printf("%d -- %d == %d\n", u, v, weight);

}

}

printf("Total weight of MST: %d\n", mstWeight);

}

int main() {

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

scanf("%d %d", &numVertices, &numEdges);

printf("Enter edges (u, v, weight):\n");

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

scanf("%d %d %d", &edges[i].u, &edges[i].v, &edges[i].weight);

}

kruskal();

return 0;

}