Experiment No.1(a)

**Aim:** Write a C program to implement operations on an Array, such as Traversing.

**Program Code: -**

#include <stdio.h>

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

printf("Array Elements:\n");

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

printf("%d ", arr[i]); // Print each element

}

printf("\n");

}

int main() {

int arr[] = {10, 20, 30, 40, 50}; // Example array

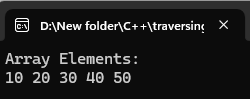
int size = sizeof(arr) / sizeof(arr[0]); // Calculate the size of the array

traverseArray(arr, size); // Call the function to traverse and print array elements

return 0;

}

**Output:-**

****

Experiment No.1(b)

**Aim:** Write a C program to implement operations on an Array, such as Insertion.

**Program Code: -**

#include <stdio.h>

int main() {

int array[5] = {1, 2, 3, 5}; // Array with initial elements

int n = 4; // Size of the current array

int element = 6; // Element to be added

int position = 4; // Position to insert the element (1-based index)

int size = sizeof(array) / sizeof(array[0]); // Calculate the size of the array

// Loop through the array and print each element

printf("Array before insertion: ");

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

printf("%d ", array[i]); // Print each element

}

printf("\n");

// Shift elements to the right to make room for the new element

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

array[i] = array[i - 1]; // Shift element to the right

}

// Insert the new element at the specified position

array[position - 1] = element;

// Increase the size of the array

n++;

// Print the array after insertion

printf("Array after insertion: ");

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

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

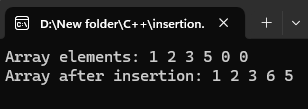
}

printf("\n");

return 0;

}

**Output:-**

****

Experiment No.1(c)

**Aim:** Write a C program to implement operations on an Array, such as Deletion.

**Program Code: -**

#include <stdio.h>

void deleteElement(int arr[], int \*size, int index) {

if (index < 0 || index >= \*size) {

printf("Invalid index!\n");

return;

}

for (int i = index; i < \*size - 1; i++) {

arr[i] = arr[i + 1]; // Shift elements to the left

}

(\*size)--; // Decrease the array size

}

int main() {

int arr[10] = {10, 20, 30, 40, 50}; // Initial array

int size = 5; // Initial size of the array

int index = 2; // Position at which to delete (0-based index)

// Loop through the array and print each element

printf("Array elements: ");

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

printf("%d ", arr[i]); // Print each element

}

printf("\n");

deleteElement(arr, &size, index);

// Print array after deletion

printf("Array after deletion:");

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

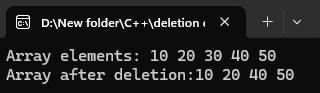
printf("%d ", arr[i]);

}

return 0;

}

**Output:-**

****

Experiment No.2(a)

**Aim:** Write a c program based on recursion such as the towers of Hanoi.

**Program Code: -**

#include <stdio.h>

// Function to solve the Towers of Hanoi problem

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

if (n == 1) {

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

return;

}

// Move n-1 disks from source to auxiliary, using destination as auxiliary

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

// Move the nth disk from source to destination

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

// Move the n-1 disks from auxiliary to destination, using source as auxiliary

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

}

int main() {

int n;

// Input number of disks

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

scanf("%d", &n);

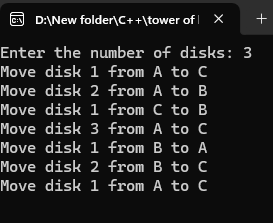
// Call the function with 3 rods: source, auxiliary, and destination

towersOfHanoi(n, 'A', 'B', 'C');

return 0;

}

**Output:-**

****

Experiment No.2(b)

**Aim:** Write a c program based on recursion such as the Fibonacci series.

**Program Code: -**

#include <stdio.h>

// Function to find the nth Fibonacci number using recursion

int fibonacci(int n) {

if (n <= 1) {

return n;

} else {

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

}

}

int main() {

int n;

// Input the number of terms for Fibonacci series

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

scanf("%d", &n);

// Print the Fibonacci series up to n terms

printf("Fibonacci Series: ");

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

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

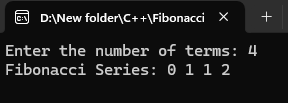
}

printf("\n");

return 0;

}

**Output:-**

****

Experiment No.3

**Aim:** Write a c program to implement a single-linked list.

**Program Code: -**

#include <stdio.h>

#include <stdlib.h>

// Define the node structure

typedef struct node {

int data;

struct node \*next;

} node;

node\* getnode();

void createlist(int n);

void displaylist(node \*start);

void deletenode(int value);

node \*start = NULL; // Global pointer to start of list

int main() {

int n, value;

printf("Enter number of nodes: ");

scanf("%d", &n);

createlist(n);

printf("\nOriginal List:\n");

displaylist(start);

printf("\nEnter value to delete: ");

scanf("%d", &value);

deletenode(value);

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

displaylist(start);

return 0;

}

void createlist(int n) {

node \*newnode, \*temp;

int i;

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

newnode = getnode();

printf("Enter data for node %d: ", i + 1);

scanf("%d", &newnode->data);

newnode->next = NULL;

if (start == NULL) {

start = newnode;

} else {

temp = start;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newnode;

}

}

}

node\* getnode() {

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

if (newnode == NULL) {

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

exit(1);

}

return newnode;

}

void displaylist(node \*start) {

node \*temp = start;

printf("Linked list: ");

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

void deletenode(int value) {

node \*temp = start, \*prev = NULL;

// If the head node itself holds the value

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

start = temp->next;

free(temp);

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

return;

}

// Search for the node to be deleted

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

prev = temp;

temp = temp->next;

}

// If not found

if (temp == NULL) {

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

return;

}

// Unlink the node and free it

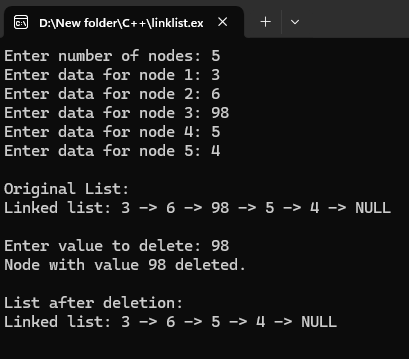
prev->next = temp->next;

free(temp);

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

}

**Output:-**



Experiment No.4

**Aim:** Write a c program to implement a double-linked list.

**Program Code: -**

#include <stdio.h>

#include <stdlib.h>

// Define a structure for the doubly linked list 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));

if (!newNode) {

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

exit(1);

}

newNode->data = data;

newNode->next = NULL;

newNode->prev = NULL;

return newNode;

}

// Function to insert a node at the beginning

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

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

} else {

newNode->next = \*head;

(\*head)->prev = newNode;

\*head = newNode;

}

}

// Function to insert a node at the end

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

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

} else {

struct Node\* temp = \*head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

}

// Function to display the list from head to tail

void displayForward(struct Node\* head) {

if (head == NULL) {

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

return;

}

struct Node\* temp = head;

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

// Function to display the list from tail to head

void displayBackward(struct Node\* head) {

if (head == NULL) {

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

return;

}

struct Node\* temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

while (temp != NULL) {

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

temp = temp->prev;

}

printf("NULL\n");

}

int main() {

struct Node\* head = NULL;

// Inserting nodes

insertAtBeginning(&head, 10);

insertAtBeginning(&head, 20);

insertAtEnd(&head, 30);

insertAtEnd(&head, 40);

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

displayForward(head);

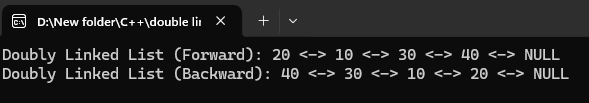
printf("Doubly Linked List (Backward): ");

displayBackward(head);

return 0;

}

**Output:-**

****

Experiment No.5

**Aim:** Write a c program to implement a circular single-linked list.

**Program Code: -**

#include <stdio.h>

#include <stdlib.h>

// Define a structure for the node in the circular linked list

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));

if (!newNode) {

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

exit(1);

}

newNode->data = data;

newNode->next = newNode; // Point to itself (circular)

return newNode;

}

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

void insertAtEnd(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->next = \*head;

}

}

// Function to display the circular linked list

void displayList(struct Node\* head) {

if (head == NULL) {

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

return;

}

struct Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head); // Continue until we get back to the head

printf("(head)\n");

}

int main() {

struct Node\* head = NULL;

// Inserting nodes into the circular linked list

insertAtEnd(&head, 10);

insertAtEnd(&head, 20);

insertAtEnd(&head, 30);

insertAtEnd(&head, 40);

// Display the circular linked list

printf("Circular Singly Linked List: ");

displayList(head);

return 0;

}

**Output:-**



Experiment No.6

**Aim:** Write a c program to implement stack operations.

**Program Code: -**

**#**include <stdio.h>

#include <stdlib.h>

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

// Define a structure for the stack

struct Stack {

int arr[MAX];

int top;

};

// Function to initialize the stack

void initStack(struct Stack\* stack) {

stack->top = -1; // Stack is initially empty

}

// Function to check if the stack is full

int isFull(struct Stack\* stack) {

return stack->top == MAX - 1;

}

// Function to check if the stack is empty

int isEmpty(struct Stack\* stack) {

return stack->top == -1;

}

// Function to push an element onto the stack

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

if (isFull(stack)) {

printf("Stack Overflow! Cannot push %d\n", value);

} else {

stack->arr[++stack->top] = value; // Increment top and insert the value

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

}

}

// Function to pop an element from the stack

int pop(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack Underflow! Cannot pop\n");

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

} else {

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

}

}

// Function to peek the top element of the stack without popping

int peek(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack is empty! Cannot peek\n");

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

} else {

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

}

}

// Function to display the stack elements

void display(struct Stack\* stack) {

if (isEmpty(stack)) {

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

} else {

printf("Stack elements: ");

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

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

}

printf("\n");

}

}

int main() {

struct Stack stack;

initStack(&stack); // Initialize the stack

// Stack operations

push(&stack, 10);

push(&stack, 20);

push(&stack, 30);

push(&stack, 40);

push(&stack, 50);

// Try to push when the stack is full

push(&stack, 60);

display(&stack);

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

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

display(&stack);

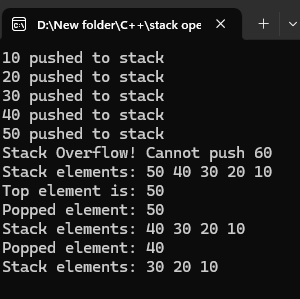
printf("Popped element: %d\n", pop(&stack));

display(&stack);

return 0;

}

**Output:-**



Experiment No.7

**Aim:** Write a c program to convert an infix expression to postfix Expression.

**Program Code: -**

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h> // For isdigit()

#define MAX 100 // Define maximum size of the stack

// Stack structure

struct Stack {

int top;

char arr[MAX];

};

// Function to initialize the stack

void initStack(struct Stack\* stack) {

stack->top = -1;

}

// Function to check if the stack is empty

int isEmpty(struct Stack\* stack) {

return stack->top == -1;

}

// Function to check if the stack is full

int isFull(struct Stack\* stack) {

return stack->top == MAX - 1;

}

// Function to push an element to the stack

void push(struct Stack\* stack, char value) {

if (isFull(stack)) {

printf("Stack Overflow\n");

return;

}

stack->arr[++stack->top] = value;

}

// Function to pop an element from the stack

char pop(struct Stack\* stack) {

if (isEmpty(stack)) {

return -1; // Return -1 if stack is empty

}

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

}

// Function to peek the top element of the stack

char peek(struct Stack\* stack) {

if (isEmpty(stack)) {

return -1; // Return -1 if stack is empty

}

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

}

// Function to check the precedence of operators

int precedence(char op) {

if (op == '+' || op == '-') {

return 1;

} else if (op == '\*' || op == '/') {

return 2;

} else if (op == '^') {

return 3;

}

return 0;

}

// Function to check if a character is an operator

int isOperator(char c) {

return (c == '+' || c == '-' || c == '\*' || c == '/' || c == '^');

}

// Function to convert infix expression to postfix

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

struct Stack stack;

initStack(&stack);

int j = 0; // Index for postfix expression

for (int i = 0; infix[i] != '\0'; i++) {

char current = infix[i];

if (isalnum(current)) {

// If the character is an operand, add it to the postfix expression

postfix[j++] = current;

}

else if (current == '(') {

// If the character is '(', push it to the stack

push(&stack, current);

}

else if (current == ')') {

// If the character is ')', pop from the stack until '(' is found

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

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

}

pop(&stack); // Pop '(' from the stack

}

else if (isOperator(current)) {

// If the character is an operator

while (!isEmpty(&stack) && precedence(peek(&stack)) >= precedence(current)) {

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

}

push(&stack, current); // Push the current operator to the stack

}

}

// Pop any remaining operators from the stack and add to postfix expression

while (!isEmpty(&stack)) {

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

}

postfix[j] = '\0'; // Null-terminate the postfix expression

}

int main() {

char infix[MAX], postfix[MAX];

// Input infix expression

printf("Enter an infix expression: ");

fgets(infix, MAX, stdin);

// Convert infix to postfix

infixToPostfix(infix, postfix);

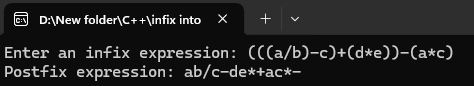
// Output postfix expression

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

return 0;

}

**Output:-**



Experiment No.8

**Aim:** Write a c program to implement queue operations.

**Program Code: -**

#include <stdio.h>

#include <stdlib.h>

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

// Define a structure for the queue

struct Queue {

int arr[MAX];

int front;

int rear;

};

// Function to initialize the queue

void initQueue(struct Queue\* queue) {

queue->front = -1; // Queue is initially empty

queue->rear = -1;

}

// Function to check if the queue is empty

int isEmpty(struct Queue\* queue) {

return queue->front == -1;

}

// Function to check if the queue is full

int isFull(struct Queue\* queue) {

return queue->rear == MAX - 1;

}

// Function to enqueue (insert) an element

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

if (isFull(queue)) {

printf("Queue Overflow! Cannot enqueue %d\n", value);

} else {

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

queue->front = 0; // If the queue was empty, set the front to 0

}

queue->arr[++queue->rear] = value; // Insert the value and move rear pointer

printf("%d enqueued to queue\n", value);

}

}

// Function to dequeue (remove) an element

int dequeue(struct Queue\* queue) {

if (isEmpty(queue)) {

printf("Queue Underflow! Cannot dequeue\n");

return -1; // Return -1 if the queue is empty

} else {

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

// If there's only one element left, reset the queue

if (queue->front == queue->rear) {

queue->front = queue->rear = -1;

} else {

queue->front++; // Move front pointer

}

return value;

}

}

// Function to peek (get the front element)

int peek(struct Queue\* queue) {

if (isEmpty(queue)) {

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

return -1; // Return -1 if the queue is empty

} else {

return queue->arr[queue->front];

}

}

// Function to display the queue elements

void display(struct Queue\* queue) {

if (isEmpty(queue)) {

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

} else {

printf("Queue elements: ");

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

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

}

printf("\n");

}

}

int main() {

struct Queue queue;

initQueue(&queue); // Initialize the queue

// Queue operations

enqueue(&queue, 10);

enqueue(&queue, 20);

enqueue(&queue, 30);

enqueue(&queue, 40);

enqueue(&queue, 50);

// Try to enqueue when the queue is full

enqueue(&queue, 60);

display(&queue);

printf("Front element is: %d\n", peek(&queue));

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

display(&queue);

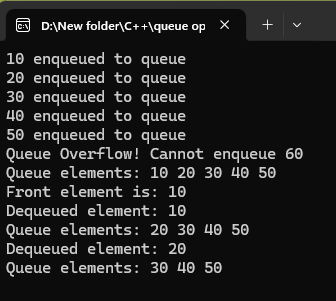
printf("Dequeued element: %d\n", dequeue(&queue));

display(&queue);

return 0;

}

**Output:-**

****

Experiment No.9

**Aim:** Write a c program to implement Tree Traversal.

**Program Code: -**

#include <stdio.h>

#include <stdlib.h>

// Define the structure of a binary tree node

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

// Function to create a new node

struct Node\* createNode(int data) {

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

if (!newNode) {

printf("Memory allocation failed\n");

exit(1);

}

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

// Inorder traversal (Left, Root, Right)

void inorderTraversal(struct Node\* root) {

if (root == NULL) return;

inorderTraversal(root->left);

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

inorderTraversal(root->right);

}

// Preorder traversal (Root, Left, Right)

void preorderTraversal(struct Node\* root) {

if (root == NULL) return;

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

preorderTraversal(root->left);

preorderTraversal(root->right);

}

// Postorder traversal (Left, Right, Root)

void postorderTraversal(struct Node\* root) {

if (root == NULL) return;

postorderTraversal(root->left);

postorderTraversal(root->right);

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

}

int main() {

// Manually create the binary tree:

//

// 1

// / \

// 2 3

// / \ \

// 4 5 6

struct Node\* root = createNode(1);

root->left = createNode(2);

root->right = createNode(3);

root->left->left = createNode(4);

root->left->right = createNode(5);

root->right->right = createNode(6);

printf("Inorder Traversal: ");

inorderTraversal(root);

printf("\n");

printf("Preorder Traversal: ");

preorderTraversal(root);

printf("\n");

printf("Postorder Traversal: ");

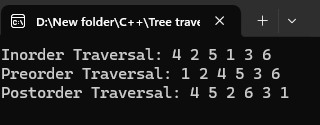
postorderTraversal(root);

printf("\n");

return 0;

}

**Output:-**

****

Experiment No.10

**Aim:** Write a c program to implement Insertion Sort.

**Program 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]; // The element to be inserted

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--;

}

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

}

}

// Function to print an array

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

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

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

}

printf("\n");

}

int main() {

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

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

printf("Original array:\n");

printArray(arr, n);

insertionSort(arr, n);

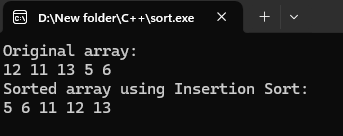
printf("Sorted array using Insertion Sort:\n");

printArray(arr, n);

return 0;

}

**Output:-**

****

Experiment No.11

**Aim:** Write a c program to implement Binary search.

**Program Code: -**

#include <stdio.h>

// Function to perform binary search

int binarySearch(int arr[], int size, int key) {

int low = 0, high = size - 1;

while (low <= high) {

int mid = (low + high) / 2;

if (arr[mid] == key)

return mid; // Key found at index mid

else if (arr[mid] < key)

low = mid + 1; // Search in right half

else

high = mid - 1; // Search in left half

}

return -1; // Key not found

}

int main() {

int arr[] = {2, 4, 6, 8, 10, 12, 14}; // Must be sorted

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

int key;

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

scanf("%d", &key);

int result = binarySearch(arr, size, key);

if (result != -1)

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

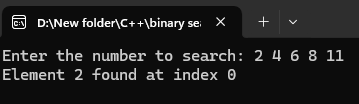
else

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

return 0;

}

**Output:-**

****

Experiment No.12

**Aim:** Write a c program to implement a Graph.

**Program Code: -**

#include <stdio.h>

#include <stdlib.h>

// Define a node in the adjacency list

struct Node {

int vertex;

struct Node\* next;

};

// Define the graph structure

struct Graph {

int numVertices;

struct Node\*\* adjLists;

};

// Function to create a new adjacency list node

struct Node\* createNode(int vertex) {

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

newNode->vertex = vertex;

newNode->next = NULL;

return newNode;

}

// Function to create a graph with a given number of vertices

struct Graph\* createGraph(int vertices) {

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

graph->numVertices = vertices;

// Allocate memory for an array of adjacency lists

graph->adjLists = (struct Node\*\*)malloc(vertices \* sizeof(struct Node\*));

// Initialize all adjacency lists as empty

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

graph->adjLists[i] = NULL;

}

return graph;

}

// Function to add an edge (for undirected graph)

void addEdge(struct Graph\* graph, int src, int dest) {

// Add edge from src to dest

struct Node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

// Add edge from dest to src (since the graph is undirected)

newNode = createNode(src);

newNode->next = graph->adjLists[dest];

graph->adjLists[dest] = newNode;

}

// Function to print the graph

void printGraph(struct Graph\* graph) {

for (int v = 0; v < graph->numVertices; v++) {

struct Node\* temp = graph->adjLists[v];

printf("Vertex %d:", v);

while (temp) {

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

temp = temp->next;

}

printf("\n");

}

}

// Free memory

void freeGraph(struct Graph\* graph) {

for (int i = 0; i < graph->numVertices; i++) {

struct Node\* current = graph->adjLists[i];

while (current != NULL) {

struct Node\* temp = current;

current = current->next;

free(temp);

}

}

free(graph->adjLists);

free(graph);

}

int main() {

int vertices = 5;

// Create a graph with 5 vertices

struct Graph\* graph = createGraph(vertices);

// Add edges

addEdge(graph, 0, 1);

addEdge(graph, 0, 4);

addEdge(graph, 1, 2);

addEdge(graph, 1, 3);

addEdge(graph, 1, 4);

addEdge(graph, 2, 3);

addEdge(graph, 3, 4);

// Print the graph

printGraph(graph);

// Free memory

freeGraph(graph);

return 0;

}

**Output:-**

