10	Write a program in C to implement circular queue using array.		
11	Write a program in C to implement circular queue using linked list.		
12	Write a program in C to implement BFS using linked list.		
13	Write a program in C to implement DFS using linked list.		
14	Write a program in C to implement Tower of Hanoi.		
15	Write a program in C to implement binary search tree using linked list.		
16	Write a program in C to implement tree traversal using linked list.		
17	Write a program in C to implement Merge Sort.		
18	Write a program in C to implement Graph traversal.		

Student Name **Student Signature**

PROGRAM-10

OBJECTIVE: Write a program in C to implement circular queue using array.

CODE:

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 5 // Maximum size of the circular queue

typedef struct {
    int items[MAX];
    int front;
    int rear;
} CircularQueue;

// Function to initialize the circular queue
void initQueue(CircularQueue* q) {
    q->front = -1;
    q->rear = -1;
}

// Function to check if the queue is full
int isFull(CircularQueue* q) {
    return (q->front == (q->rear + 1) % MAX);
}

// Function to check if the queue is empty
int isEmpty(CircularQueue* q) {
    return (q->front == -1);
}

// Function to add an element to the queue
void enqueue(CircularQueue* q, int value) {
    if (isFull(q)) {
        printf("Queue is full! Cannot enqueue %d\n", value);
        return;
    }
    if (isEmpty(q)) {
        q->front = 0; // Set front to 0 when first element is added
    }
    q->rear = (q->rear + 1) % MAX; // Circular increment
    q->items[q->rear] = value;
    printf("Enqueued: %d\n", value);
}

// Function to remove an element from the queue
int dequeue(CircularQueue* q) {
```

```
    if (isEmpty(q)) {
        printf("Queue is empty! Cannot dequeue\n");
        return -1; // Indicate that the queue is empty
    }
    int item = q->items[q->front];
    if (q->front == q->rear) {
        // Queue has only one element, reset queue
        q->front = -1;
        q->rear = -1;
    } else {
        q->front = (q->front + 1) % MAX; // Circular increment
    }
    printf("Dequeued: %d\n", item);
    return item;
}

// Function to display the elements of the queue
void display(CircularQueue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Queue elements: ");
    int i = q->front;
    while (1) {
        printf("%d ", q->items[i]);
        if (i == q->rear) {
            break;
        }
        i = (i + 1) % MAX; // Circular increment
    }
    printf("\n");
}

int main() {
    CircularQueue q;
    initQueue(&q);

    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);
    display(&q);

    dequeue(&q);
    display(&q);

    enqueue(&q, 40);
```

```
    enqueue(&q, 50);
    display(&q);

    enqueue(&q, 60); // This should show that the queue is full
    dequeue(&q);
    display(&q);

    return 0;
}
```

OUTPUT:

```
Enqueued: 10
Enqueued: 20
Enqueued: 30
Queue elements: 10 20 30
Dequeued: 10
Queue elements: 20 30
Enqueued: 40
Enqueued: 50
Queue elements: 20 30 40 50
Enqueued: 60
Dequeued: 20
Queue elements: 30 40 50 60


...Program finished with exit code 0
Press ENTER to exit console.
```

PROGRAM-11

OBJECTIVE: Write a program in C to implement circular queue using linked list.

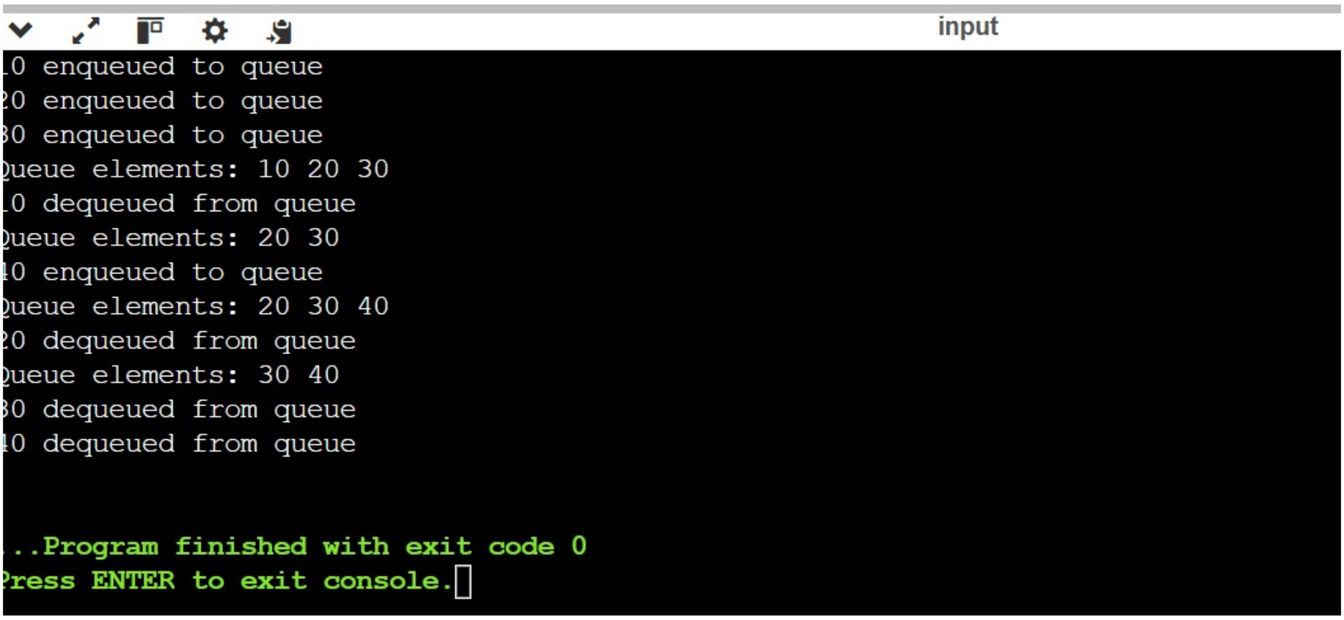
CODE:

```
#include <stdio.h>
#include <stdlib.h>

// Node structure
struct Node {
    int data;
    struct Node* next;
};

// Circular Queue structure
struct CircularQueue {
    struct Node* front;
    struct Node* rear;
};

// 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 = NULL;
    return newNode;
}

// Function to create a circular queue
struct CircularQueue* createQueue() {
    struct CircularQueue* queue = (struct CircularQueue*)malloc(sizeof(struct CircularQueue));
    queue->front = queue->rear = NULL;
    return queue;
}

// Function to check if the queue is empty
int isEmpty(struct CircularQueue* queue) {
    return (queue->front == NULL);
}

// Function to enqueue an element
void enqueue(struct CircularQueue* queue, int value) {
    struct Node* newNode = createNode(value);
    if (isEmpty(queue)) {
        queue->front = queue->rear = newNode;
        newNode->next = newNode; // Point to itself
    } else {
        queue->rear->next = newNode;
        queue->rear = newNode;
        queue->rear->next = queue->front; // Maintain circularity
    }
    printf("%d enqueued to queue\n", value);
}

// Function to dequeue an element
```

```c
int dequeue(struct CircularQueue* queue) {
	if (isEmpty(queue)) {
		printf("Queue is empty, cannot dequeue\n");
		return -1; // Indicating that the queue is empty
	}
	int value = queue->front->data;
	if (queue->front == queue->rear) {
		free(queue->front);
		queue->front = queue->rear = NULL; // Queue becomes empty
	} else {
		struct Node* temp = queue->front;
		queue->front = queue->front->next;
		queue->rear->next = queue->front; // Maintain circularity
		free(temp);
	}
	printf("%d dequeued from queue\n", value);
	return value;
}

// Function to display the queue
void display(struct CircularQueue* queue) {
	if (isEmpty(queue)) {
		printf("Queue is empty\n");
		return;
	}
	struct Node* temp = queue->front;
	printf("Queue elements: ");
	do {
		printf("%d ", temp->data);
		temp = temp->next;
	} while (temp != queue->front);
	printf("\n");
}

// Main function to test the circular queue
int main() {
	struct CircularQueue* queue = createQueue();

	enqueue(queue, 10);
	enqueue(queue, 20);
	enqueue(queue, 30);
	display(queue);

	dequeue(queue);
	display(queue);

	enqueue(queue, 40);
	display(queue);

	dequeue(queue);
	display(queue);

	// Clean up remaining elements
	while (!isEmpty(queue)) {
		dequeue(queue);
```

```
	}
	free(queue);

	return 0;
}
```

OUTPUT:



PROGRAM-12

OBJECTIVE: Write a program in C to implement BFS using linked list.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAX 100

// Structure to represent a node in the adjacency list
struct Node {
	int vertex;
	struct Node* next;
};

// Structure to represent the graph
struct Graph {
	int numVertices;
	struct Node** adjLists;
};

// Queue structure for BFS
struct Queue {
	int items[MAX];
	int front;
	int rear;
};

// Function to create a new adjacency list node
struct Node* createNode(int v) {
	struct Node* newNode = malloc(sizeof(struct Node));
	newNode->vertex = v;
	newNode->next = NULL;
	return newNode;
}

// Function to create a graph
struct Graph* createGraph(int vertices) {
	struct Graph* graph = malloc(sizeof(struct Graph));
	graph->numVertices = vertices;

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

	for (int i = 0; i < vertices; i++) {
		graph->adjLists[i] = NULL;
	}

	return graph;
}

// Function to add an edge to the 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 (for undirected graph)
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// Function to create a queue
struct Queue* createQueue() {
    struct Queue* q = malloc(sizeof(struct Queue));
    q->front = -1;
    q->rear = -1;
    return q;
}

// Function to check if the queue is empty
bool isEmpty(struct Queue* q) {
    return q->rear == -1;
}

// Function to add an item to the queue
void enqueue(struct Queue* q, int value) {
    if (q->rear == MAX - 1) {
        printf("Queue is full\n");
    } else {
        if (q->front == -1) {
            q->front = 0;
        }
        q->rear++;
        q->items[q->rear] = value;
    }
}

// Function to remove an item from the queue
int dequeue(struct Queue* q) {
    int item;
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return -1;
    } else {
        item = q->items[q->front];
        q->front++;
        if (q->front > q->rear) {
            q->front = q->rear = -1; // Reset the queue
        }
        return item;
    }
}

// BFS algorithm
void bfs(struct Graph* graph, int startVertex) {
```

```
    bool visited[MAX] = {false}; // Array to track visited vertices
    struct Queue* q = createQueue();

    visited[startVertex] = true; // Mark the starting vertex as visited
    enqueue(q, startVertex); // Enqueue the starting vertex
    while (!isEmpty(q)) {
        int currentVertex = dequeue(q);
        printf("Visited %d\n", currentVertex);

        struct Node* temp = graph->adjLists[currentVertex];
        while (temp) {
            int adjVertex = temp->vertex;
            if (!visited[adjVertex]) {
                visited[adjVertex] = true; // Mark as visited
                enqueue(q, adjVertex); // Enqueue the adjacent vertex
            }
            temp = temp->next;
        }
    }
}
int main() {
    struct Graph* graph = createGraph(5); // Create a graph with 5 vertices
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 4);
    printf("BFS starting from vertex 0:\n");
    bfs(graph, 0);
    return 0;
}
```

OUTPUT:

PROGRAM-13

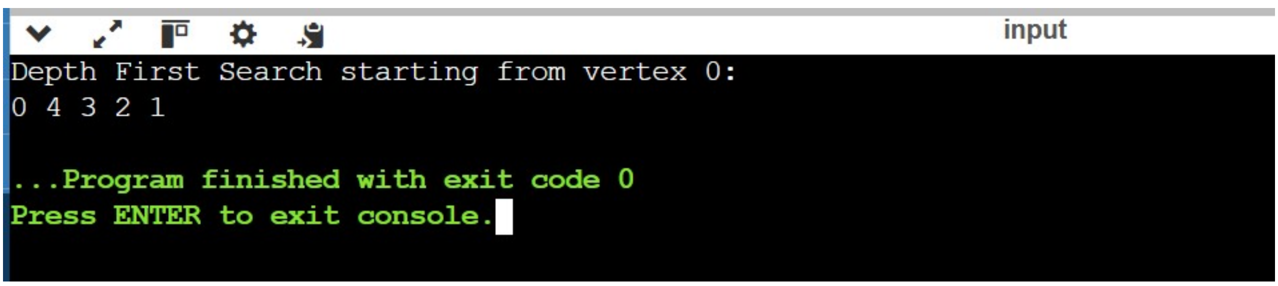
OBJECTIVE: Write a program in C to implement DFS using linked list.

CODE:

```
#include <stdio.h>
#include <stdlib.h>

// Define a structure for a node in the adjacency list
struct Node {
	int vertex;
	struct Node* next;
};

// Define a structure for the graph
struct Graph {
	int numVertices;
	struct Node** adjLists;
	int* visited;
};

// Function to create a new adjacency list node
struct Node* createNode(int v) {
	struct Node* newNode = malloc(sizeof(struct Node));
	newNode->vertex = v;
	newNode->next = NULL;
	return newNode;
}

// Function to create a graph with a given number of vertices
struct Graph* createGraph(int vertices) {
	struct Graph* graph = malloc(sizeof(struct Graph));
	graph->numVertices = vertices;

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

	for (int i = 0; i < vertices; i++) {
		graph->adjLists[i] = NULL;
		graph->visited[i] = 0; // Initialize all vertices as not visited
	}

	return graph;
}

// Function to add an edge to the 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;

	// Since the graph is undirected, add an edge from dest to src
	newNode = createNode(src);
```

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

// DFS algorithm
void DFS(struct Graph* graph, int vertex) {
	// Mark the current node as visited and print it
	graph->visited[vertex] = 1;
	printf("%d ", vertex);

	// Recur for all the vertices adjacent to this vertex
	struct Node* temp = graph->adjLists[vertex];
	while (temp) {
		int adjVertex = temp->vertex;
		if (!graph->visited[adjVertex]) {
			DFS(graph, adjVertex);
		}
		temp = temp->next;
	}
}

int main() {
	struct Graph* graph = createGraph(5); // Create a graph with 5 vertices
	// Add edges to the graph
	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);
	printf("Depth First Search starting from vertex 0:\n");
	DFS(graph, 0);

	return 0;
}
```

OUTPUT:



PROGRAM-14

OBJECTIVE: Write a program in C to implement Tower of Hanoi.

CODE:

```c
#include <stdio.h>

// Function to perform the Tower of Hanoi algorithm
void towerOfHanoi(int n, char source, char destination, char auxiliary) {
    // Base case: If there's only one disk, move it directly from source to destination
    if (n == 1) {
        printf("Move disk 1 from peg %c to peg %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 peg %c to peg %c\n", n, source, destination);

    // Move the n-1 disks from auxiliary peg to destination peg
    towerOfHanoi(n - 1, auxiliary, destination, source);
}

int main() {
    int n; // Number of disks
    printf("Enter the number of disks: ");
    scanf("%d", &n);
  // Call the function to solve the Tower of Hanoi
    towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of the rods
    return 0;
}
```

OUTPUT:

input

```
Enter the number of disks: 3
Move disk 1 from peg A to peg C
Move disk 2 from peg A to peg B
Move disk 1 from peg C to peg B
Move disk 3 from peg A to peg C
Move disk 1 from peg B to peg A
Move disk 2 from peg B to peg C
Move disk 1 from peg A to peg C


...Program finished with exit code 0
Press ENTER to exit console.
```

PROGRAM-15

OBJECTIVE: Write a program in C to implement binary search tree using linked list.

CODE:

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for a 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));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Function to insert a new node in the BST
struct Node* insert(struct Node* root, int data) {
    if (root == NULL) {
        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;
}

// Function to search for a value in the BST
struct Node* search(struct Node* root, int data) {
    if (root == NULL || root->data == data) {
        return root;
    }
    if (data < root->data) {
        return search(root->left, data);
    }
    return search(root->right, data);
}

// Function for in-order traversal of the BST
void inOrderTraversal(struct Node* root) {
    if (root != NULL) {
        inOrderTraversal(root->left);
        printf("%d ", root->data);
```

```
        inOrderTraversal(root->right);
    }
}

// Main function to demonstrate the BST
int main() {
    struct Node* root = NULL;
    int choice, value;

    do {
        printf("\nBinary Search Tree Operations:\n");
        printf("1. Insert\n");
        printf("2. Search\n");
        printf("3. In-order Traversal\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                root = insert(root, value);
                break;
            case 2:
                printf("Enter value to search: ");
                scanf("%d", &value);
                struct Node* result = search(root, value);
                if (result != NULL) {
                    printf("Value %d found in the BST.\n", value);
                } else {
                    printf("Value %d not found in the BST.\n", value);
                }
                break;
            case 3:
                printf("In-order Traversal: ");
                inOrderTraversal(root);
                printf("\n");
                break;
            case 4:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice! Please try again.\n");
        }
    } while (choice != 4);

    return 0;
}
```

OUTPUT:

input

```
Binary Search Tree Operations:
1. Insert
2. Search
3. In-order Traversal
4. Exit
Enter your choice: 1
Enter value to insert: 10

Binary Search Tree Operations:
1. Insert
2. Search
3. In-order Traversal
4. Exit
Enter your choice: 1
Enter value to insert: 2
```

input

```
Binary Search Tree Operations:
1. Insert
2. Search
3. In-order Traversal
4. Exit
Enter your choice: 1
Enter value to insert: 4

Binary Search Tree Operations:
1. Insert
2. Search
3. In-order Traversal
4. Exit
Enter your choice: 3
In-order Traversal: 2 4 10

Binary Search Tree Operations:
1. Insert
2. Search
3. In-order Traversal
4. Exit
Enter your choice: 4
Exiting...
```

PROGRAM-16

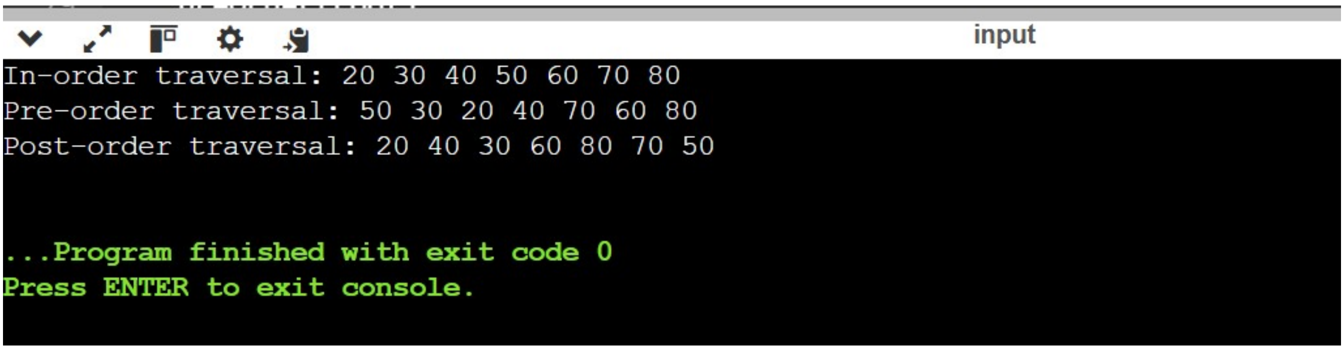
OBJECTIVE: Write a program in C to implement tree traversal using linked list.

CODE:

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for a tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new tree node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Function to insert a new node in the binary tree
struct Node* insert(struct Node* node, int data) {
    if (node == NULL) {
        return createNode(data);
    }
    if (data < node->data) {
        node->left = insert(node->left, data);
    } else {
        node->right = insert(node->right, data);
    }
    return node;
}

// Function for in-order traversal
void inOrder(struct Node* root) {
    if (root != NULL) {
        inOrder(root->left);
        printf("%d ", root->data);
        inOrder(root->right);
    }
}

// Function for pre-order traversal
void preOrder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preOrder(root->left);
        preOrder(root->right);
    }
```

```c
}

// Function for post-order traversal
void postOrder(struct Node* root) {
	if (root != NULL) {
		postOrder(root->left);
		postOrder(root->right);
		printf("%d ", root->data);
	}
}

// Main function to demonstrate tree traversal
int main() {
	struct Node* root = NULL;
	// Insert nodes into the binary tree
	root = insert(root, 50);
	insert(root, 30);
	insert(root, 20);
	insert(root, 40);
	insert(root, 70);
	insert(root, 60);
	insert(root, 80);
	// Display the tree traversals
	printf("In-order traversal: ");
	inOrder(root);
	printf("\n");
	printf("Pre-order traversal: ");
	preOrder(root);
	printf("\n");
	printf("Post-order traversal: ");
	postOrder(root);
	printf("\n");
	return 0;
}
```

OUTPUT:



PROGRAM-17

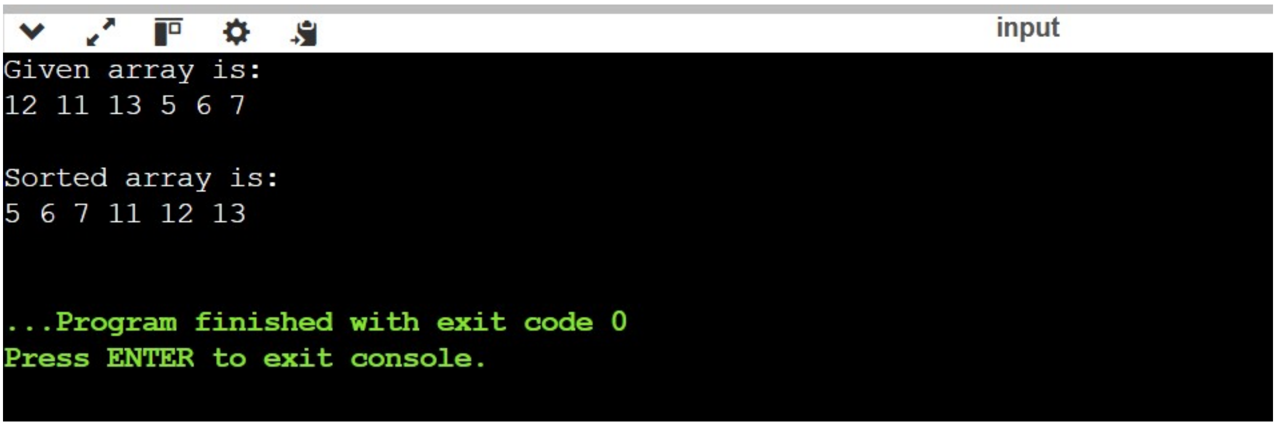
OBJECTIVE: Write a program in C to implement Merge Sort.

CODE:

```
#include <stdio.h>

// Function to merge two halves of an array
void merge(int arr[], int left, int mid, int right) {
	int i, j, k;
	int n1 = mid - left + 1; // Size of the left subarray
	int n2 = right - mid;	// Size of the right subarray

	// 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[left + i];
	for (j = 0; j < n2; j++)
		R[j] = arr[mid + 1 + j];

	// Merge the temporary arrays back into arr[left..right]
	i = 0; // Initial index of first subarray
	j = 0; // Initial index of second subarray
	k = left; // 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 left, int right) {
```

```
	if (left < right) {
		// Find the middle point
		int mid = left + (right - left) / 2;

		// Sort first and second halves
		mergeSort(arr, left, mid);
		mergeSort(arr, mid + 1, right);

		// Merge the sorted halves
		merge(arr, left, mid, right);
	}
}

// Function to print an array
void printArray(int arr[], int size) {
	for (int i = 0; i < size; i++)
		printf("%d ", arr[i]);
	printf("\n");
}

// Main function
int main() {
	int arr[] = {12, 11, 13, 5, 6, 7};
	int arr_size = sizeof(arr) / sizeof(arr[0]);

	printf("Given array is: \n");
	printArray(arr, arr_size);

	mergeSort(arr, 0, arr_size - 1);

	printf("\nSorted array is: \n");
	printArray(arr, arr_size);
	return 0;
}
```

OUTPUT:



PROGRAM-18

OBJECTIVE: Write a program in C to implement Graph traversal.

CODE:

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_VERTICES 100

typedef struct Node {
	int vertex;
	struct Node* next;
} Node;

typedef struct Graph {
	Node* adjLists[MAX_VERTICES];
	int visited[MAX_VERTICES];
	int numVertices;
} Graph;

// Function to create a new adjacency list node
Node* createNode(int v) {
	Node* newNode = malloc(sizeof(Node));
	newNode->vertex = v;
	newNode->next = NULL;
	return newNode;
}

// Function to initialize the graph
Graph* createGraph(int vertices) {
	Graph* graph = malloc(sizeof(Graph));
	graph->numVertices = vertices;

	for (int i = 0; i < vertices; i++) {
		graph->adjLists[i] = NULL;
		graph->visited[i] = 0; // Initialize all vertices as unvisited
	}
	return graph;
}

// Function to add an edge to the graph
void addEdge(Graph* graph, int src, int dest) {
	// Add edge from src to dest
	Node* newNode = createNode(dest);
	newNode->next = graph->adjLists[src];
	graph->adjLists[src] = newNode;

	// For undirected graph, add edge from dest to src
	newNode = createNode(src);
	newNode->next = graph->adjLists[dest];
	graph->adjLists[dest] = newNode;
}
```

```
// Depth-First Search (DFS) algorithm
void DFS(Graph* graph, int vertex) {
	graph->visited[vertex] = 1;
	printf("%d ", vertex);

	Node* adjList = graph->adjLists[vertex];
	while (adjList != NULL) {
		int connectedVertex = adjList->vertex;
		if (graph->visited[connectedVertex] == 0) {
			DFS(graph, connectedVertex);
		}
		adjList = adjList->next;
	}
}

// Breadth-First Search (BFS) algorithm
void BFS(Graph* graph, int startVertex) {
	int queue[MAX_VERTICES];
	int front = 0;
	int rear = -1;

	graph->visited[startVertex] = 1;
	printf("%d ", startVertex);
	queue[++rear] = startVertex;

	while (front <= rear) {
		int currentVertex = queue[front++];

		Node* adjList = graph->adjLists[currentVertex];
		while (adjList != NULL) {
			int connectedVertex = adjList->vertex;

			if (graph->visited[connectedVertex] == 0) {
				graph->visited[connectedVertex] = 1;
				printf("%d ", connectedVertex);
				queue[++rear] = connectedVertex;
			}
			adjList = adjList->next;
		}
	}
}

// Main function to demonstrate graph traversal
int main() {
	int vertices = 5;
	Graph* graph = createGraph(vertices);

	// Adding edges to the graph
	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);

    printf("DFS Traversal starting from vertex 0:\n");
    DFS(graph, 0);
    printf("\n");

    // Resetting visited array for BFS
    for (int i = 0; i < vertices; i++) {
        graph->visited[i] = 0;
    }

    printf("BFS Traversal starting from vertex 0:\n");
    BFS(graph, 0);
    printf("\n");

    // Free allocated memory (not shown for simplicity)
    // In production code, you should free the allocated memory.

    return 0;
}
```

OUTPUT:

input

```
DFS Traversal starting from vertex 0:
0 4 3 2 1
BFS Traversal starting from vertex 0:
0 4 1 3 2


...Program finished with exit code 0
Press ENTER to exit console.
```