EXPERIMENT –10 [Implementation of Graphs]

1. **For a given graph G=(V,E), study and implement the Breadth First Search. Also, perform the complexity analysis of this algorithm in terms of time and space.**

SOURCE CODE:

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

struct queue
{
  int size;
  int f;
  int r;
  int* arr;
};
int isEmpty(struct queue *q){
  if(q->r==q->f){
    return 1;
  }
  return 0;
}
int isFull(struct queue *q){
  if(q->r==q->size-1){
    return 1;
  }
  return 0;
}

void enqueue(struct queue *q, int val){
  if(isFull(q)){
    printf("This Queue is full\n");
  }
  else{
    q->r++;
    q->arr[q->r] = val;
    // printf("Enqued element: %d\n", val);
  }
}

int dequeue(struct queue *q){
  int a = -1;
  if(isEmpty(q)){
    printf("This Queue is empty\n");
  }
  else{
    q->f++;
    a = q->arr[q->f];
  }
  return a;
}

int main(){
  struct queue q;
  q.size = 400;
  q.f = q.r = 0;
  q.arr = (int*) malloc(q.size*sizeof(int));

  int node;
  int i = 1;
  int visited[7] = {0,0,0,0,0,0,0};
  int a [7][7] = {
    {0,1,1,1,0,0,0},
    {1,0,1,0,0,0,0},
    {1,1,0,1,1,0,0},
    {1,0,1,0,1,0,0},
    {0,0,1,1,0,1,1},
    {0,0,0,0,1,0,0},
    {0,0,0,0,1,0,0}
  };
  printf("%d", i);
  visited[i] = 1;
  enqueue(&q, i); // Enqueue i for exploration
  while (!isEmpty(&q))
  {
    int node = dequeue(&q);
    for (int j = 0; j < 7; j++)
    {
        if(a[node][j] ==1 && visited[j] == 0){
            printf("%d", j);
            visited[j] = 1;
            enqueue(&q, j);
        }
    }
  }
  return 0;
}
```

OUTPUT:

```
0123456
```

2)For a given graph G=(V,E), study and implement the Depth First Search. Also, perform the complexity analysis of this algorithm in terms of time and space.

SOURCE CODE:

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

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

struct Graph {
   int V;        // Number of vertices
   struct Node** adjList; // Array of adjacency lists
};

struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = NULL;
   return newNode;
}

struct Graph* createGraph(int V) {
   struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
   graph->V = V;
   graph->adjList = (struct Node**)malloc(V * sizeof(struct Node*));

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

   return graph;
}

void addEdge(struct Graph* graph, int src, int dest) {
   struct Node* newNode = createNode(dest);
   newNode->next = graph->adjList[src];
   graph->adjList[src] = newNode;
}

void DFS(struct Graph* graph, int vertex, int* visited) {
   visited[vertex] = 1;
   printf("%d ", vertex);

   struct Node* current = graph->adjList[vertex];
```

```
  while (current != NULL) {
    int neighbor = current->data;
    if (!visited[neighbor]) {
      DFS(graph, neighbor, visited);
    }
    current = current->next;
  }
}

int main() {
  int V = 6; // Number of vertices in the graph
  struct Graph* graph = createGraph(V);

  // Add edges to the graph
  addEdge(graph, 0, 1);
  addEdge(graph, 0, 2);
  addEdge(graph, 1, 3);
  addEdge(graph, 2, 4);
  addEdge(graph, 3, 5);

  int* visited = (int*)malloc(V * sizeof(int));

  for (int i = 0; i < V; i++) {
    visited[i] = 0; // Initialize the visited array
  }
  printf("DFS Traversal starting from vertex 0: ");
  DFS(graph, 0, visited);

  free(visited);

  return 0;
}
```

OUTPUT:

```
DFS Traversal starting from vertex 0: 0 1 3 5 2 4
```

3) Given a directed graph, check whether the graph contains a cycle or not. Your function should return true if the given graph contains at least one cycle, else false. Perform same task for undirected graph as well.

SOURCE CODE:

=>FOR DIRECTED GRAPH

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

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

#define MAX_VERTICES 100

struct Graph {
    int V;
    int** adjMatrix;
};

bool isCyclicUtil(struct Graph* graph, int v, bool* visited, bool* recursionStack) {
    if (!visited[v]) {
        visited[v] = true;
        recursionStack[v] = true;

        for (int i = 0; i < graph->V; i++) {
            if (graph->adjMatrix[v][i] == 1) {
                if (!visited[i] && isCyclicUtil(graph, i, visited, recursionStack)) {
                    return true;
                }
                else if (recursionStack[i]) {
                    return true;
                }
            }
        }
    }

    recursionStack[v] = false;
    return false;
}

bool hasCycle(struct Graph* graph) {
    bool* visited = (bool*)calloc(graph->V, sizeof(bool));
    bool* recursionStack = (bool*)calloc(graph->V, sizeof(bool));

    for (int i = 0; i < graph->V; i++) {
        if (!visited[i] && isCyclicUtil(graph, i, visited, recursionStack)) {
            free(visited);
            free(recursionStack);
            return true;
        }
    }

    free(visited);
    free(recursionStack);
    return false;
}
```

```
int main() {
  int V = 4;
  struct Graph graph;
  graph.V = V;
  graph.adjMatrix = (int**)malloc(V * sizeof(int*));

  for (int i = 0; i < V; i++) {
    graph.adjMatrix[i] = (int*)calloc(V, sizeof(int));
  }

  graph.adjMatrix[0][1] = 1;
  graph.adjMatrix[1][2] = 1;
  graph.adjMatrix[2][3] = 1;
  graph.adjMatrix[3][1] = 1;

  if (hasCycle(&graph)) {
    printf("The directed graph contains at least one cycle.\n");
  } else {
    printf("The directed graph does not contain a cycle.\n");
  }

  for (int i = 0; i < V; i++) {
    free(graph.adjMatrix[i]);
  }
  free(graph.adjMatrix);

  return 0;
}
```

OUTPUT:

```
The directed graph contains at least one cycle.
```

FOR UNDIRECTED GRAPH:

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

#define MAX_VERTICES 100

struct Graph {
  int V;
  int** adjMatrix;
};

bool isCyclicUtil(struct Graph* graph, int v, int parent, bool* visited) {
    visited[v] = true;

    for (int i = 0; i < graph->V; i++) {
        if (graph->adjMatrix[v][i] == 1) {
            if (!visited[i]) {
                if (isCyclicUtil(graph, i, v, visited)) {
                    return true;
                }
            }
            else if (i != parent) {
                return true;
            }
        }
    }

    return false;
}

bool hasCycle(struct Graph* graph) {
    bool* visited = (bool*)calloc(graph->V, sizeof(bool));

    for (int i = 0; i < graph->V; i++) {
        if (!visited[i] && isCyclicUtil(graph, i, -1, visited)) {
            free(visited);
            return true;
        }
    }

    free(visited);
    return false;
}

int main() {
    int V = 4; // Number of vertices in the undirected graph
    struct Graph graph;
    graph.V = V;
    graph.adjMatrix = (int**)malloc(V * sizeof(int*));

    for (int i = 0; i < V; i++) {
        graph.adjMatrix[i] = (int*)calloc(V, sizeof(int));
    }

    // Define the adjacency matrix for the undirected graph
    graph.adjMatrix[0][1] = 1;
    graph.adjMatrix[1][2] = 1;
    graph.adjMatrix[2][3] = 1;
    graph.adjMatrix[3][0] = 1;
    if (hasCycle(&graph)) {
        printf("The undirected graph contains at least one cycle.\n");
    } else {
        printf("The undirected graph does not contain a cycle.\n");
    }

    for (int i = 0; i < V; i++) {
        free(graph.adjMatrix[i]);
    }
    free(graph.adjMatrix);

    return 0;
}
```

OUTPUT:

```
The undirected graph contains at least one cycle.
```