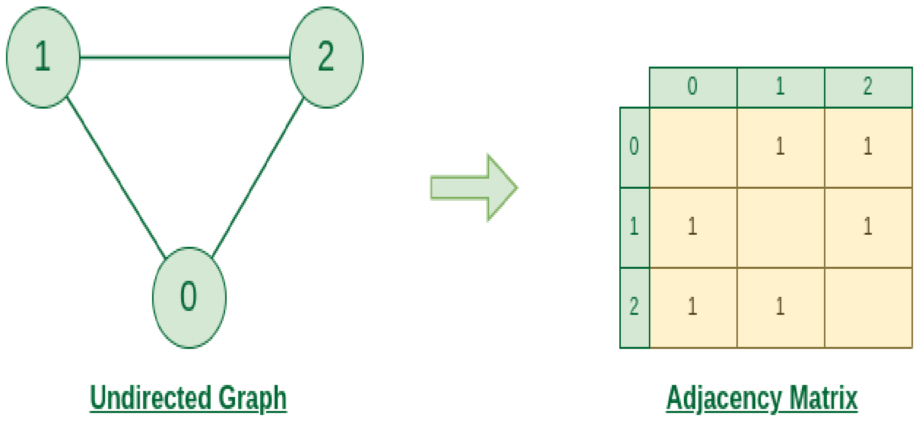
Name	Maureen Miranda
UID no.	2022300060
Experiment No.	8

AIM:	To implement graphs using adjacency matrix
Program 1	
PROBLEM STATEMENT :	-> Implement BFS & DFS traversal for graphs. -> You need to make use of an adjacency matrix for representing the graph
THEORY:	An adjacency matrix is a way of representing a graph as a matrix of boolean (0's and 1's). Let's assume there are n vertices in the graph So, create a 2D matrix adjMat[n][n] having dimension n x n. If there is an edge from vertex i to j, mark adjMat[i][j] as 1. If there is no edge from vertex i to j, mark adjMat[i][j] as 0. Representation of Undirected Graph to Adjacency Matrix: The below figure shows an undirected graph. Initially, the entire Matrix is initialized to 0. If there is an edge from source to destination, we insert 1 to both cases (adjMat[destination] and adjMat[destination]) because we can go either way.



	0	1	2
0		1	1
1	1		1
2	1	1	

Graph Representation of Undirected graph to Adjacency Matrix

Representation of Directed Graph to Adjacency Matrix:
. Initially, the entire Matrix is initialized to 0. If there is an edge from source to destination, we insert 1 for that particular adjMat[destination].

Breadth-First Traversal (or Search) for a graph is similar to the Breadth-First Traversal of a tree.

The only catch here is, that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

Visited and
Not visited.
A boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a queue data structure for traversal

Depth First Traversal (or DFS) for a graph is similar to Depth First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a boolean visited array. A graph can have more than one DFS traversal.

Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

Algorithm:

Graph Initialization Algorithm (`init_graph`)

1. Create a function `init_graph` that takes the number of vertices (`num_vertices`) and a boolean flag indicating if the graph is directed (`is_directed`) as parameters.
2. Allocate memory for a `GraphRep` structure and store it in a pointer `graph`.
3. Allocate memory for various arrays within the `GraphRep` structure, including `edges`, `color`, `distance`, `finish`, `predecessor`.
4. Set the number of vertices (`nV`) to `num_vertices`, the number of edges (`nE`) to 0, and the `is_directed` flag as specified.
5. Initialize the `type` to `BFS`, and `source` to 0 as default values.
6. Initialize the `color` array to all `WHITE`, `distance` to 0, `finish` to 0, and `predecessor` to -1 for all vertices.
7. Initialize the adjacency matrix `edges` with `false` for all pairs of vertices.
8. Return the `graph` pointer as the initialized graph.

Insert Edge Algorithm (`insert_edge`)

1. Create a function `insert_edge` that takes a `GraphRep` pointer `graph` and an `Edge` structure `e` (with `u` and `v` denoting the vertices).
2. Get the source and destination vertices `u` and `v` from the `Edge` structure.
3. Set the `edges[u][v]` and `edges[v][u]` to `true` to represent the edge between vertices `u` and `v`.
4. If the graph is undirected, also set `edges[v][u]` to `true`.
5. Set the color of vertex `u` to `WHITE` (indicating undiscovered).
6. Increment the number of edges in the graph (`nE`) by 1.

Remove Edge Algorithm (`remove_edge`)

1. Create a function `remove_edge` that takes a `GraphRep` pointer `graph` and

```
an `Edge` structure `e`.
2. Get the source and destination vertices `u` and `v` from the `Edge` structure.
3. Set the `edges[u][v]` and `edges[v][u]` to `false` to remove the edge between vertices `u` and `v`.
4. If the graph is undirected, also set `edges[v][u]` to `false`.
5. Decrement the number of edges in the graph (`nE`) by 1.
```

Breadth-First Search Traversal Algorithm (`traverse_bfs`)

```
1. Create a function `traverse_bfs` that takes a `GraphRep` pointer `graph` and a starting vertex `source`.
2. Initialize a queue to store vertices and enqueue the `source` vertex.
3. Initialize a variable `i` to the `source` vertex.
4. Perform the following loop while the queue is not empty:
   - Dequeue the front vertex from the queue and set it as `i`.
   - Print the value of `i`.
   - Iterate through the vertices connected to `i`:
     - If an edge exists from `i` to vertex `k` and the color of `k` is `WHITE`, then:
       - Enqueue vertex `k`.
       - Set the color of `k` to `GRAY`.
       - Set the predecessor of `k` to `i`.
       - Update the distance of `k` as the distance of `i` plus 1.
   - Set the color of `i` to `BLACK`.
```

Depth-First Search Traversal Algorithm (`traverse_dfs`)

```
1. Create a function `traverse_dfs` that takes a `GraphRep` pointer `graph` and a starting vertex `source`.
2. Set the traversal type to `DFS`.
3. Set the color of the `source` vertex to `GRAY`.
4. Print the value of the `source` vertex.
5. Iterate through the vertices in the graph:
   - If there is an edge from the `source` to vertex `i`, and the color of `i` is `WHITE`, then:
     - Set the predecessor of `i` to `source`.
     - Set the finish time of `i` as the finish time of `source` plus 1.
     - Recursively call `traverse_dfs` on vertex `i`.
     - Update the finish time of `i` as the finish time of `source` plus 1.
6. Set the color of the `source` vertex to `BLACK`.
```

Display Path Algorithm (`display_path`)

1. Create a function `display_path` that takes a `GraphRep` pointer `graph` and a destination vertex.
2. Check the traversal type (`BFS` or `DFS`) in the `graph` structure.
3. Print the type of traversal used to reach the destination.
4. Print the source vertex.
5. Call a helper function (`path`) to backtrack and print the path from the `source` to the destination.

Display Path Helper Algorithm (`path`)

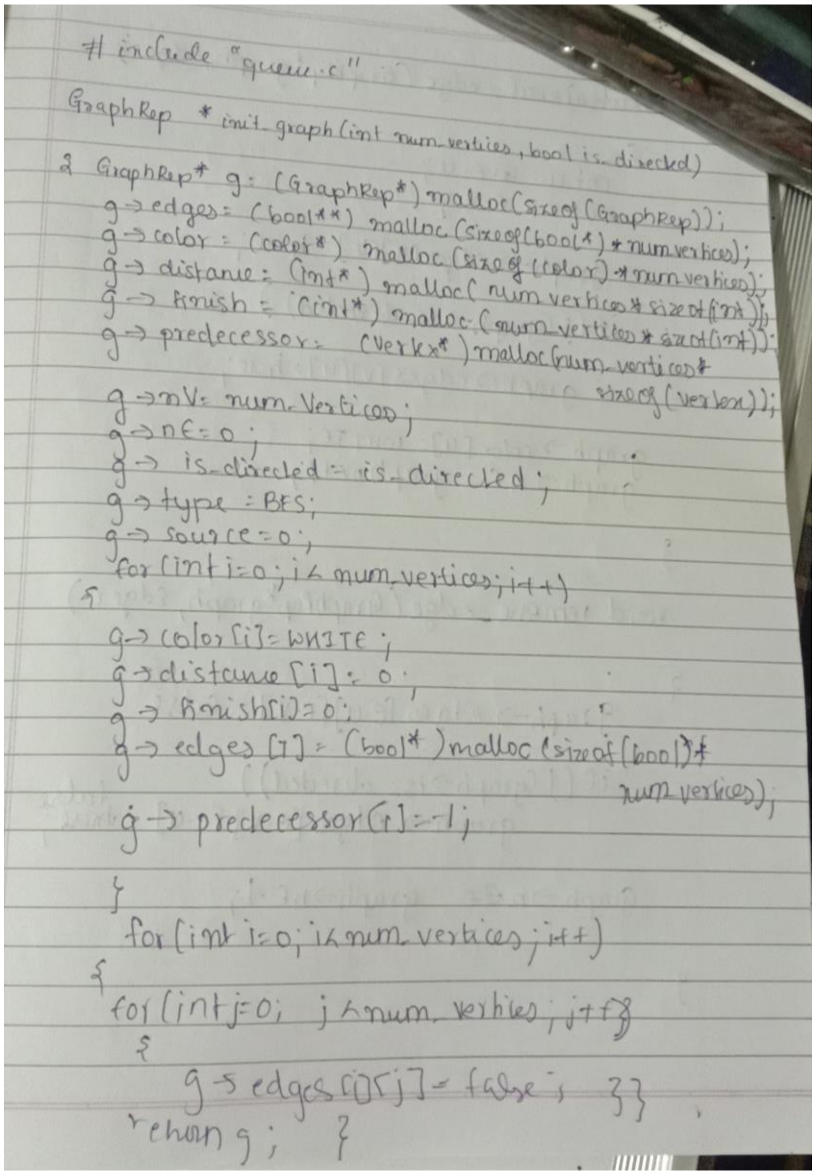
1. Create a function `path` that takes a `GraphRep` pointer `graph`, a source vertex, and a destination vertex.
2. If the source is equal to the destination, return.
3. Recursively call `path` with the source and the predecessor of the destination and then print the destination.

Display Graph Algorithm (`display_graph`)

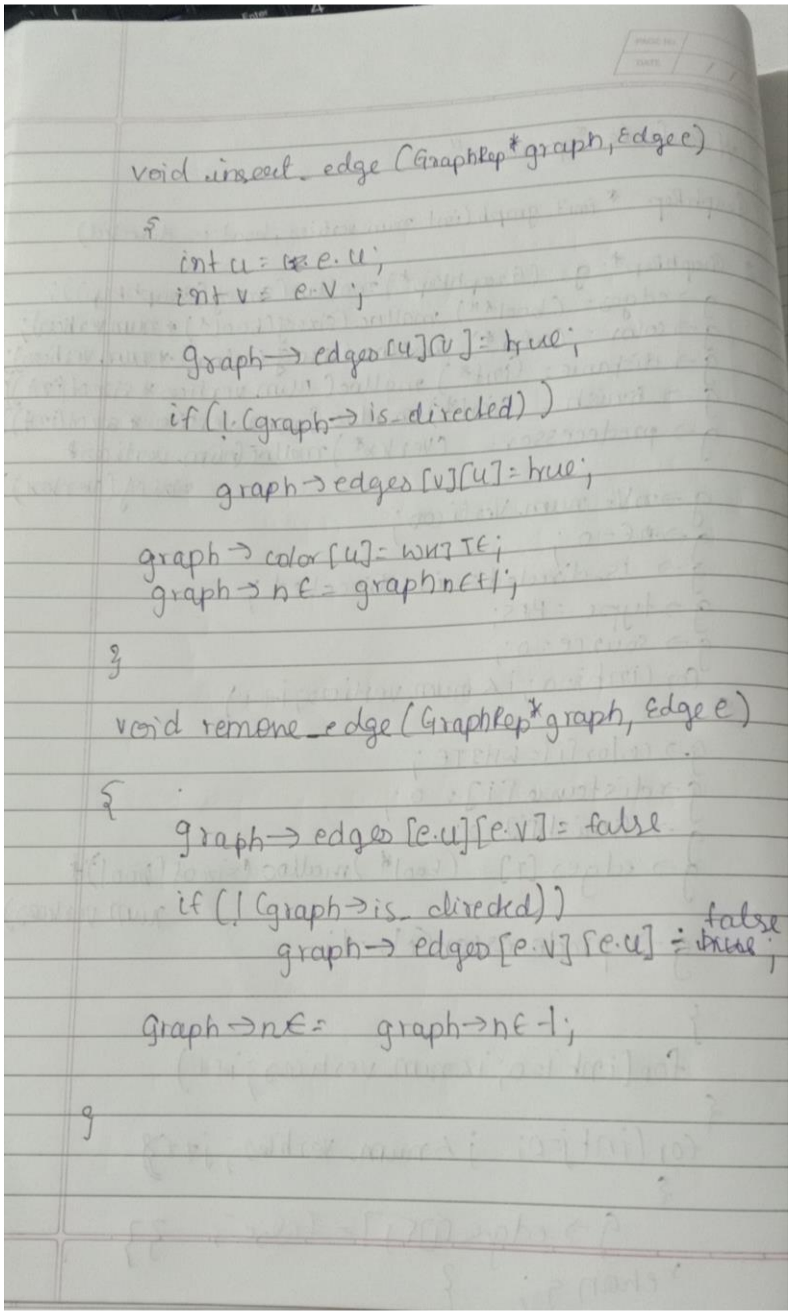
1. Create a function `display_graph` that takes a `GraphRep` pointer `graph`.
2. Print the adjacency matrix to display the graph in matrix form.

This structured algorithm outlines the key steps and operations performed by each function in the provided C code. You can use this as a reference to implement the actual logic within each function.

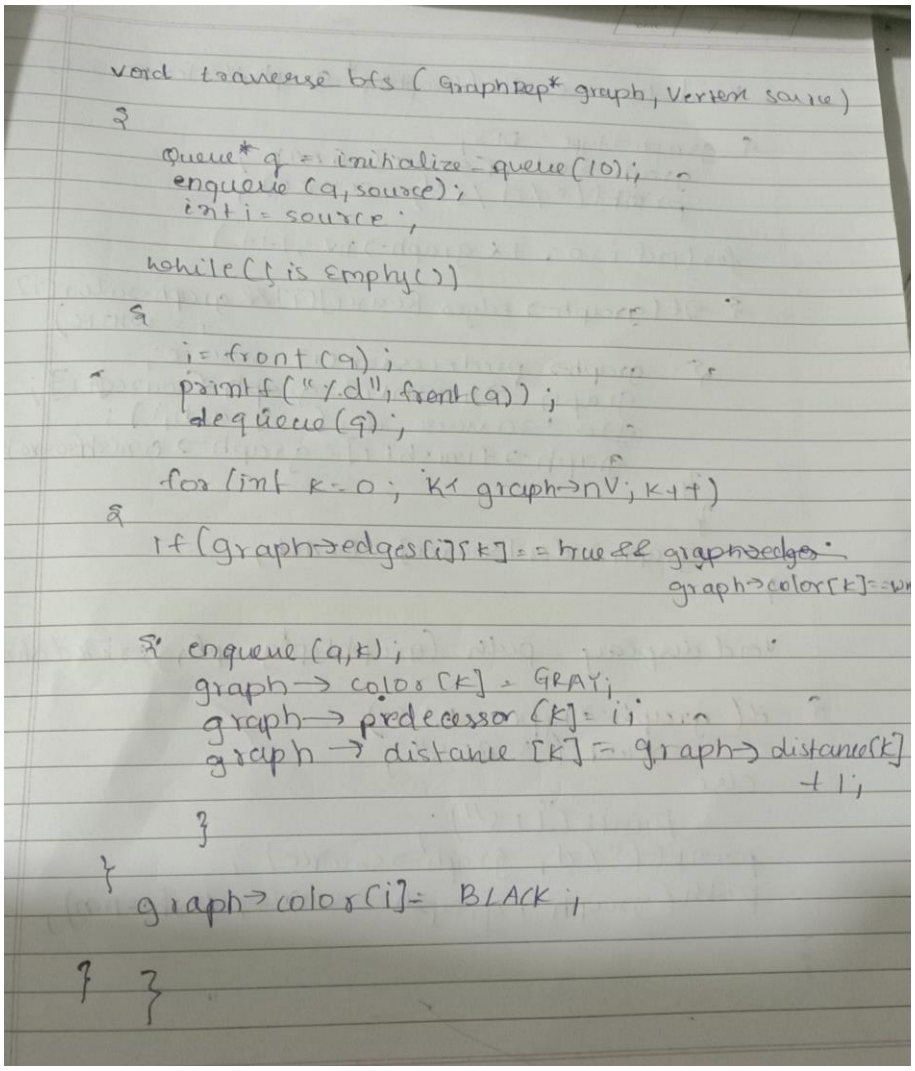
Solution :

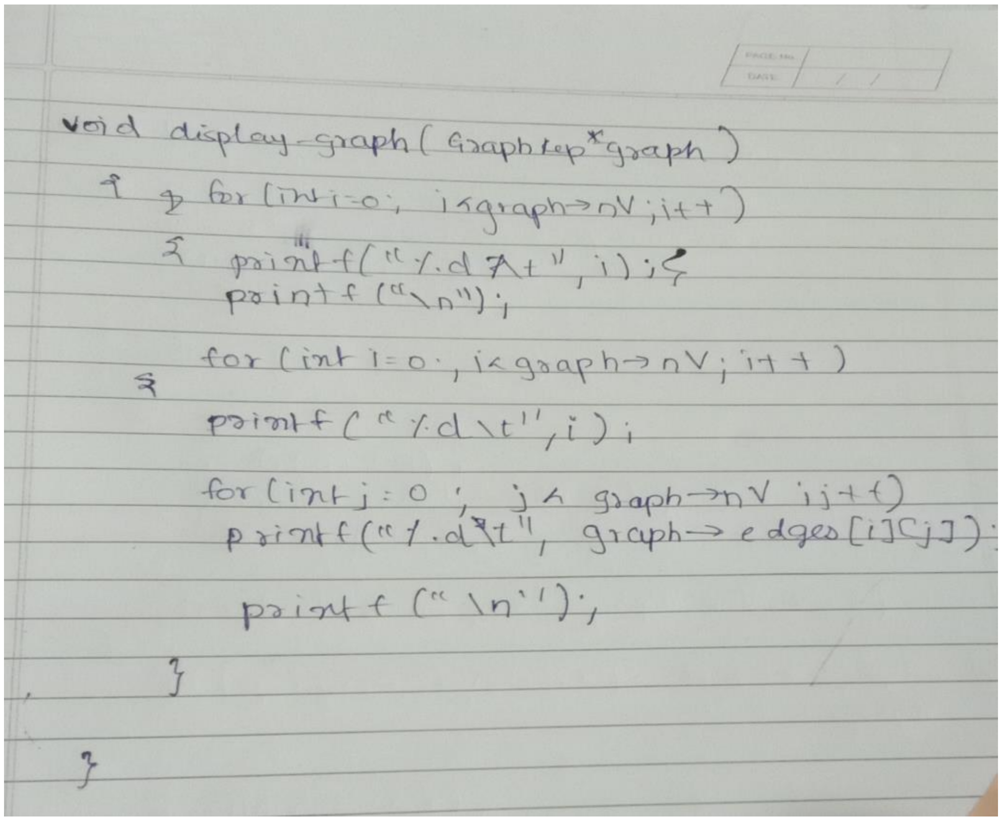
```
#include "queue.c"

GraphRep *init_graph(int num_vertices, bool is_direcked)

{ GraphRep* g= (GraphRep*) malloc(sizeof(GraphRep));
  g->edges= (bool**) malloc(sizeof(bool*) * num_vertices);
  g->color = (color*) malloc(sizeof(color) * num_vertices);
  g->distance= (int*) malloc( num vertices * sizeof(int));
  g-> finish = (int*) malloc( num_vertices * sizeof(int));
  g->predecessor= (Vertex*) malloc(num_vertices*
                                      sizeof(vertex));
  g->nV= num_Vertices;
  g->nE=0;
  g-> is_directed = is_directed;
  g->type =BFS;
  g-> source=0;
  for(int i=0; i< num_vertices; i++)
 {
  g->color[i]=WHITE;
  g->distance[i]= 0;
  g-> finish[i]=0;
  g-> edges [i] = (bool*)malloc(sizeof(bool)*
                                      num vertices);
  g-> predecessor[i]=-1;

  }
   for(int i=0; i<num_vertices; i++)
 {
  for(int j=0;  j<num_vertices; j++)
   {
       g-> edges[i][j]=false;   }}
   return g;   }
```

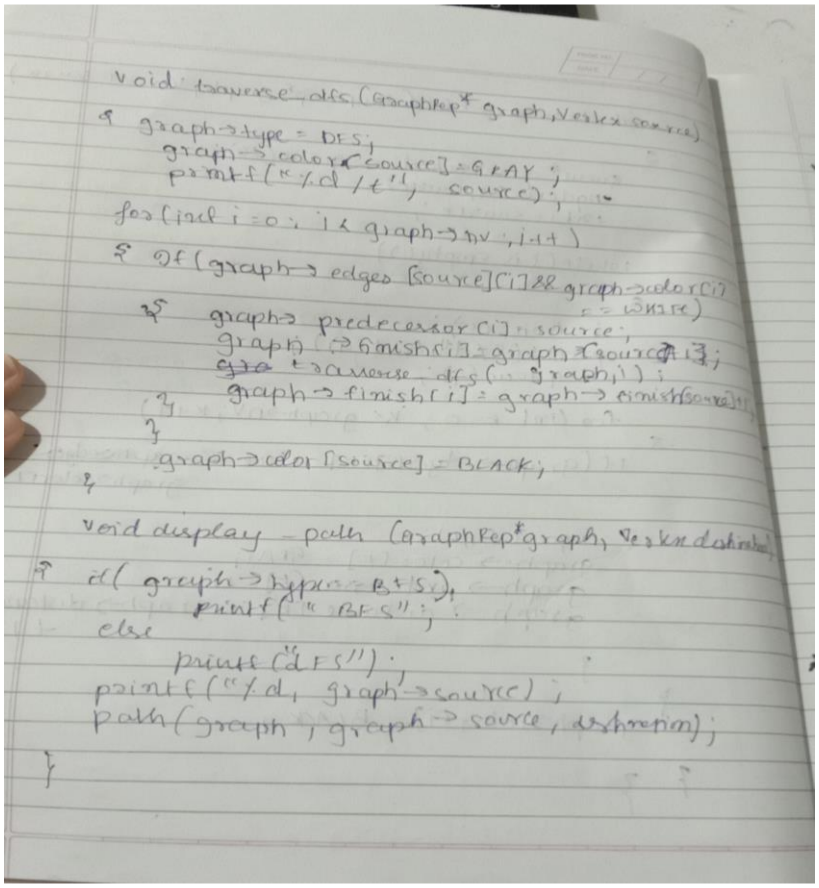
```
void insert_edge (GraphRep* graph, Edge e)
{
    int u = ~~x~~ e.u;
    int v = e.v;
    graph→ edges[u][v] = true;
    if (!(graph→is_directed))
        graph→edges[v][u] = true;
    graph→color[u] = WHITE;
    graph→nE = graphnE+1;
}
void remove_edge(GraphRep* graph, Edge e)
{
    graph→edges[e.u][e.v] = false
    if (!(graph→is_directed))
        graph→edges[e.v][e.u] = ~~true~~ false;
    graph→nE = graph→nE-1;
}
```

```
void traverse bfs ( GraphRep* graph, Vertex source)
{
   Queue* q = initialize_queue(10);
   enqueue (q, source);
   int i = source ;

  while(( is Empty())
  {
     i = front (q) ;
     printf ("%d", front (q)) ;
     dequeue (q) ;

    for (int k = 0 ; K< graph->nV; k++)
  {
    if (graph->edges[i][k] == true && ~~graph->edges~~
                                    graph->color[k]==w
   {  enqueue (q,k) ;
      graph-> color [K] = GRAY;
       graph-> predecessor [K] = i;
       graph -> distance [K] = graph-> distance[k]
                                              + 1;
     }
  }
   graph->color[i] = BLACK ;

}  }
```

```
void display_graph( Graphrep* graph )
  {  for (int i=0;  i<graph->nV; i++)
     {  printf("%d \t", i); }
        printf("\n");

        for (int i=0 ; i<graph->nV; i++ )
     {
        printf( "%d \t", i);

        for (int j=0 ;  j< graph->nV ; j++)
         printf("%d\t", graph-> edges[i][j]);

         printf ("\n");

     }

}
```



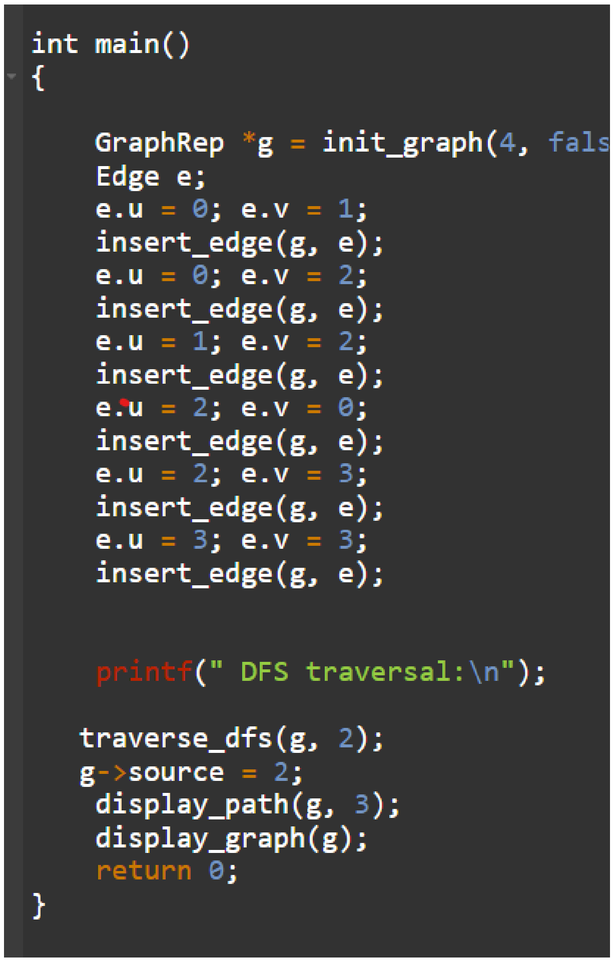
```
void traverse_dfs (GraphRep* graph, Vertex source)
{ graph->type = DFS;
   graph->color[source] = GRAY ;
   printf("%d /t", source);
for (int i = 0 ; i < graph->nv ; i++ )
{ if (graph->edges [source][i] && graph->color[i]
                                       == WHITE)
  {   graph->predecessor [i] = source ;
      graph ->finish[i] = graph->[source]+1;
      ~~gra~~ traverse_dfs ( graph, i ) ;
      graph->finish[i] = graph->finish[source]+1;
  }
  }
  graph->color [source] = BLACK;
}

void display_path (GraphRep* graph, Vertex destination)
{ if( graph->type == BFS)
        printf(" BFS");
  else
        printf("dFS");
  printf("%d, graph->source) ;
  path(graph , graph->source, destination);
}
```

PROGRAM:

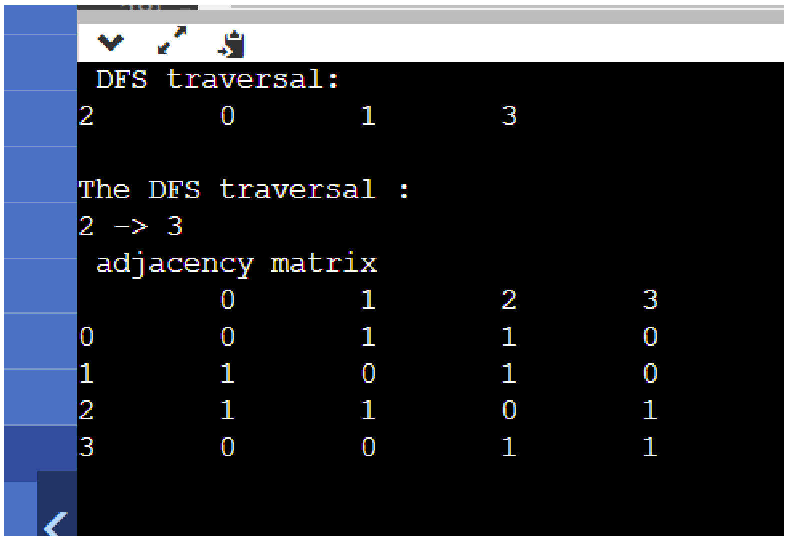
```
#include "queue.c"
GraphRep *init_graph(int num_vertices, bool is_directed)
{
GraphRep * g= (GraphRep*)malloc(sizeof(GraphRep));
g->edges=(bool**)malloc(sizeof(bool*)*num_vertices);
g->color = (Color*)malloc(num_vertices*sizeof(Color));
g->distance = (int*)malloc(num_vertices*sizeof(int));
g->finish = (int*)malloc(num_vertices*sizeof(int));
 g->predecessor = (Vertex*)malloc(num_vertices*sizeof(Vertex));
g->nV=num_vertices;
g->nE=0;
g->is_directed=is_directed;
g->type=BFS;
g->source=0;
for(int i=0;i<num_vertices;i++)
{g->color[i]=WHITE;
g->distance[i] =0;
g->finish[i]=0;
g->edges[i]=(bool*)malloc(sizeof(bool)*num_vertices);
g->predecessor[i]=-1;
}
for(int i=0;i<num_vertices;i++)
{
for(int j=0;j<num_vertices;j++)
{
g->edges[i][j]=false;
}
}

return g;
}
void insert_edge(GraphRep *graph, Edge e)
{

int u = e.u;
int v= e.v;
graph->edges[u][v]=true;
if(!(graph->is_directed))
{
    graph->edges[v][u]=true;
}
graph->color[u]=WHITE;
graph->nE=graph->nE+1;
}
void remove_edge(GraphRep *graph, Edge e)
{
int u = e.u;
int v= e.v;
graph->edges[u][v]=false;
if(!(graph->is_directed))
{
    graph->edges[v][u]=false;
}
graph->nE=graph->nE-1;
//check color;
}
// NOTE: During both DFS and BFS traversals, when at a vertex that is connected with multiple vertices, always pick the connecting vertex which has the lowest value first
// Both traversals will always update the following attributes of the Graph:
// 1. source -> stores the value of the starting vertex for the traversal
// 2. type -> stores the traversal type (BFS or DFS)
// 3. color --> indicates if all vertices have been visited or not. Post traversal, all vertices should be BLACK
// 4. predecessor --> this array would hold the predecessor for a given vertex (indicated by the array index).

// NOTE: BFS Traversal should additionally update the following in the graph:
// 1. distance --> this array would hold the number of hops it takes to reach a given vertex (indicated by the array index) from the source.
void traverse_bfs(GraphRep *graph, Vertex source)
{
     Queue *q = initialize_queue(10);
    enqueue(q,source);

    int i =source;
while(!isEmpty(q))
{
  i=front(q);
```

```
    printf("%d ",front(q));
    dequeue(q);

    for(int k=0;k<graph->nV;k++)
    {
        if(graph->edges[i][k]==true&&graph->color[k]==WHITE)
        {
            enqueue(q,k);
           graph->color[k]=GRAY;
           graph->predecessor[k]=i;
           graph->distance[k]=graph->distance[i]+1;
        }
    }
    graph->color[i]=BLACK;


}
}



// NOTE: DFS Traversal should additionally update the following in the graph:
// 1. distance --> Assuming 1 hop to equal 1 time unit, this array would hold the
time of discovery a given vertex (indicated by the array index) from the source.
// 2. finish --> Assuming 1 hop to equal 1 time unit, this array would hold the
time at which exploration concludes for a given vertex (indicated by the array
index).
void traverse_dfs(GraphRep *graph, Vertex source){
    graph->type = DFS;
    graph->color[source] = GRAY;
    printf("%d\t", source);
    for(int i=0; i<graph->nV; i++){
        if(graph->edges[source][i] && graph->color[i] == WHITE){
            graph->predecessor[i] = source;
            graph->finish[i] = graph->finish[source]+1;
            traverse_dfs(graph, i);
            graph->finish[i] = graph->finish[source]+1;
        }
    }
    graph->color[source] = BLACK;
}
```

```
// displays the path from the current 'source' in graph to the provided
'destination'.
// The graph holds the value of the traversal type, so the function should let the
caller know what kind of traversal was done on the graph and from which
vertex, along with the path.
void path(GraphRep *graph, Vertex source, Vertex destination){
    if(source==destination)
    return;
    path(graph, source, graph->predecessor[destination]);
    printf("-> %d ", destination);
}
// displays the path from the current 'source' in graph to the provided
'destination'.
// The graph holds the value of the traversal type, so the function should let the
caller know what kind of traversal was done on the graph and from which
vertex, along with the path.
void display_path(GraphRep *graph, Vertex destination){
    if(graph->type == BFS){
        printf("\n\nThe BFS traversal to reach the destination is as follows:\n");
    }
    else{
        printf("\n\nThe DFS traversal to reach the destination is as follows:\n");
    }
     printf("%d ", graph->source);
        path(graph, graph->source, destination);
}
// display the graph in the matrix form
void display_graph(GraphRep *graph)
{
 printf("\n adjacency matrix \n:");
    for(int i=0; i<graph->nV; i++)
    {printf("%d\t", i);}
 printf("\n");
    for(int i=0; i<graph->nV; i++){
        printf("%d\t", i);
        for(int j=0; j<graph->nV; j++){
            printf("%d\t", graph->edges[i][j]);
        }
        printf("\n");
    }
```

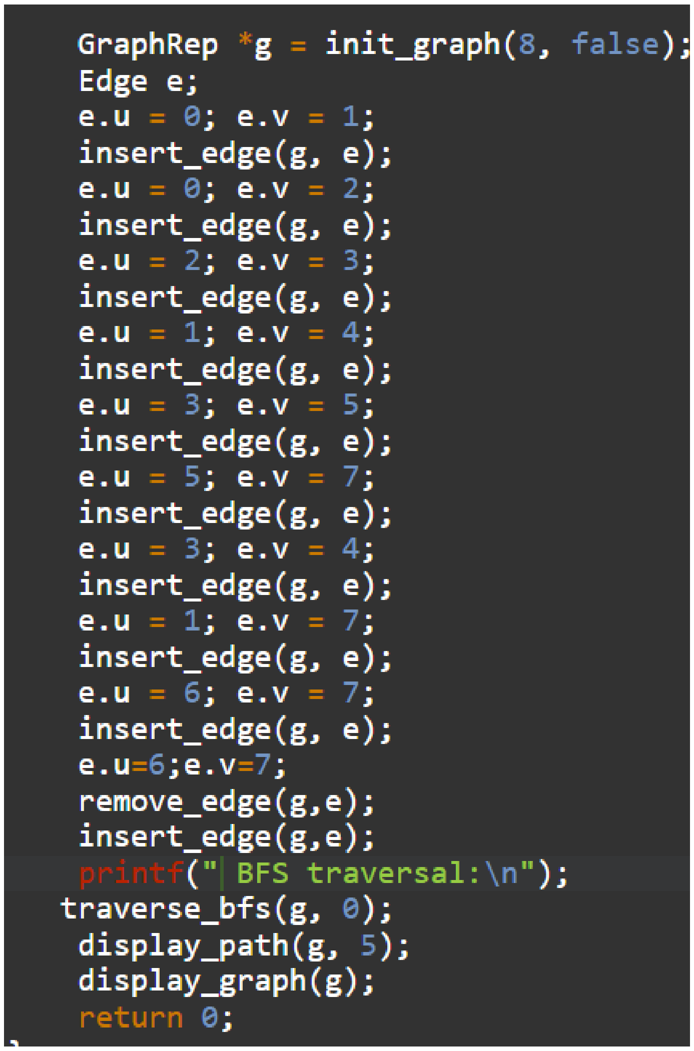
	}
RESULT:	Case 1: 

```
int main()
{

    GraphRep *g = init_graph(4, fals
    Edge e;
    e.u = 0; e.v = 1;
    insert_edge(g, e);
    e.u = 0; e.v = 2;
    insert_edge(g, e);
    e.u = 1; e.v = 2;
    insert_edge(g, e);
    e.u = 2; e.v = 0;
    insert_edge(g, e);
    e.u = 2; e.v = 3;
    insert_edge(g, e);
    e.u = 3; e.v = 3;
    insert_edge(g, e);


    printf(" DFS traversal:\n");

   traverse_dfs(g, 2);
   g->source = 2;
    display_path(g, 3);
    display_graph(g);
    return 0;
}
```



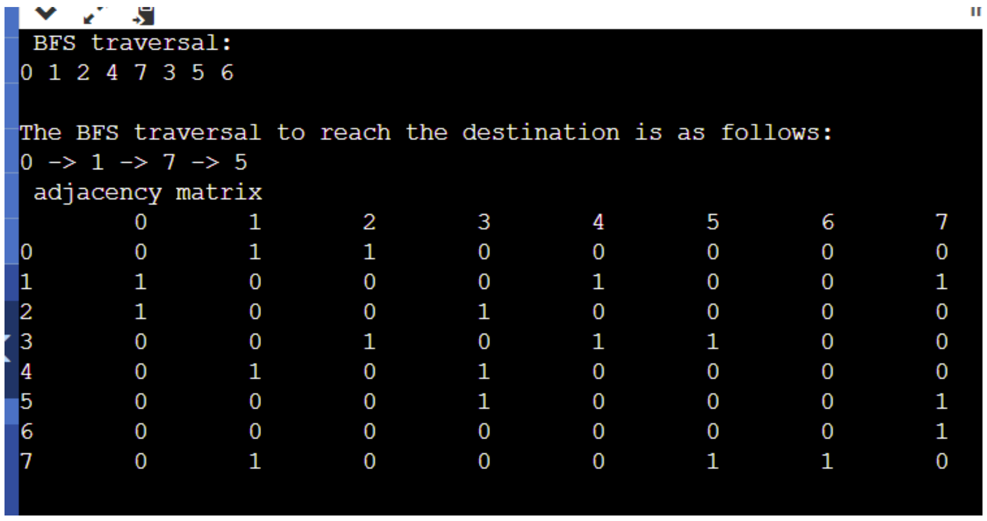
```
 DFS traversal:
2       0       1       3

The DFS traversal :
2 -> 3
 adjacency matrix
        0       1       2       3
0       0       1       1       0
1       1       0       1       0
2       1       1       0       1
3       0       0       1       1
```

Case 2:



```c
    GraphRep *g = init_graph(8, false);
    Edge e;
    e.u = 0; e.v = 1;
    insert_edge(g, e);
    e.u = 0; e.v = 2;
    insert_edge(g, e);
    e.u = 2; e.v = 3;
    insert_edge(g, e);
    e.u = 1; e.v = 4;
    insert_edge(g, e);
    e.u = 3; e.v = 5;
    insert_edge(g, e);
    e.u = 5; e.v = 7;
    insert_edge(g, e);
    e.u = 3; e.v = 4;
    insert_edge(g, e);
    e.u = 1; e.v = 7;
    insert_edge(g, e);
    e.u = 6; e.v = 7;
    insert_edge(g, e);
    e.u=6;e.v=7;
    remove_edge(g,e);
    insert_edge(g,e);
    printf(" BFS traversal:\n");
   traverse_bfs(g, 0);
    display_path(g, 5);
    display_graph(g);
    return 0;
```

Result:



```
 BFS traversal:
0 1 2 4 7 3 5 6

The BFS traversal to reach the destination is as follows:
0 -> 1 -> 7 -> 5
 adjacency matrix
        0       1       2       3       4       5       6       7
0       0       1       1       0       0       0       0       0
1       1       0       0       0       1       0       0       1
2       1       0       0       1       0       0       0       0
3       0       0       1       0       1       1       0       0
4       0       1       0       1       0       0       0       0
5       0       0       0       1       0       0       0       1
6       0       0       0       0       0       0       0       1
7       0       1       0       0       0       1       1       0
```

Conclusion :	We have learnt about the data structure graph and the way to implement it using adjacency matrix . we have also learnt 2 ways of traversal that is breadth first search and depth first search , also displaying the path of traversal and the adjacency matrix.